ADVANCED OPERATING SYSTEMS

(includes all slides for part A but not the seminars)

Davide Giannubilo a.a. 22/23

---

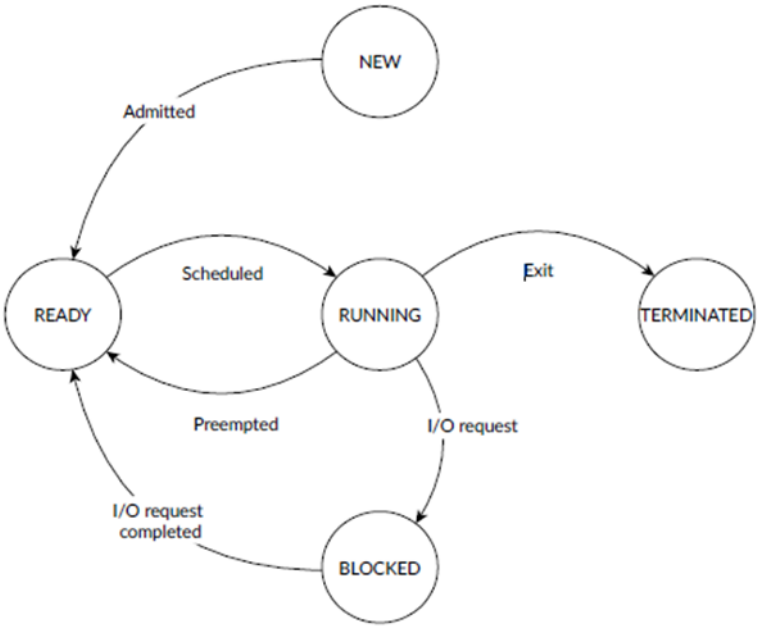
1. Introduction

1.1 Definition, goals and techniques of an OS

An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs.

- Resource management (CPU(s) and IO bandwidth)
- Isolation and protection (Memory and access control)
- Make it easier to port and extend (System call interfaces, drivers, and virtual filesystem)

The OS has a PCB (Process Control Block) for each process. It is a struct and it resides in kernel space. The purpose of this struct is to track *state* of the process and other information.



The OS must decide which process to run next (**Scheduling**) and the policy in which it decides should balance:

- Fairness – do not starve processes
- Throughput – want good overall performance
- Efficiency – minimize overhead of scheduler itself
- Priority – reflect relative importance of processes
- Deadlines – must do X by certain time

There is no universal policy.

For **isolation and protection**, it uses a *Virtual Address Space* that is the set of ranges of virtual addresses that an operating system makes available to a process.

It is built up from virtual memory areas:

- Some of them from the on-disk representation of the program (.text, .rodata, .bss, etc.)
- Others built dynamically (heap, stack, etc.)
- Other are just not accessible to processes (kernel space)

Another way to provide isolation and protection is the *Access Control Matrix* that is a graph that defines subjects that can access an object in which way.
Policy: defines the way in which this graph evolves with the state of the system.
(Traditional Unix systems uses access control lists, simplifies them by classifying subjects into roles: *owner*, *group*, *everyone*)

Design pattern: it is a typical solution to commonly occurring problems in software design. Pre-made blueprint that you can customize to solve a recurring design problem in your code.
OS system design is heavily based on pattern. The most used ones in OSes are the *facade* (ex. system call interfaces) and the *bridge* (ex. file systems or peripheral drivers) pattern.

> **System call**: an application can invoke kernel through system call. It produces an exception and transfers control to kernel which dispatches to one of few hundred syscall handlers. System calls are numbered, and the number and function parameters are set into CPU registers. A software interrupt instruction is executed to switch mode and execute the call.
>
> **File system**: the OS provides a set of mechanism and policies to regulate access to persistent storage in multiple ways and across multiple processes.
> The goal is to:
>
> - Provide abstraction
> - Regulate space usage
> - Provide protection

1.2 Architectures of OSs

The design of an operating system can follow a hybrid of different architectural approaches

- No OS at all - bare metal programming
- Monolithic with modules
  - Single large kernel binary
  - Device drivers and kernels are part of the same executable and reside in the same memory area
  - Examples: Linux, Embedded Linux, AIX, HP-UX, Solaris, *BSD
- Micro-kernel
  - The kernel includes only a subset of core components
  - Additional services are implemented via external modules
  - Modules can be dynamically linked on demand, at run-time
  - All *non-essential* components of the kernel are implemented as processes in user space

    - Due to its asynchronous nature, a crash of a system process does not mean a crash of the entire system
    - Examples: SeL4, GNU Hurd, MINIX, MkLinux, QNX and Redox OS
- Hybrid
    - Similar to micro kernels
    - Include some additional code in kernel-space to increase performance
        - Run some services (such as the network stack or the filesystem) in kernel space
        - Run device drivers in user space
    - Examples: Windows NT, 2000, XP, Vista, 7, 8, 8.1 and 10, macOS
- Library OS
    - Services such as networking are provided in the form of libraries and compiled with the application and configuration code
    - An Unikernel is a specialized, single address space, machine image that can be deployed to cloud or embedded environments (RTOSes).
    - Examples: FreeRTOS, IncludeOS, MirageOS

2. Linux processes

2.1 Tasks

A task is a program with:

- a unique program counter
- two stacks, one in user mode and one in kernel mode
- a set of processor registers, and (optional) an address space

They can be categorized as follow:

- if it shares memory with other tasks then these are called **threads** (in Linux they are *tasks*)
- otherwise, it is called a **process**

A task can be in user space of kernel space at any given moment and when it is in user space or kernel space on behalf of the process (e.g., executing a system call), we say that the kernel is in **process context**.

Kernel can create threads which live always in **kernel-mode**. They perform some operations in background, their address space is the entire kernel's one and, they are schedulable and preemptable as normal processes.

Each task is represented in Linux by a **process descriptor** (`task_struct`); these are collected in a list (**the task list**).
The kernel must be able to access this data structure without much work:

- Some architectures save a pointer to the `task_struct` structure of the currently running process **in a register**
- Other architectures, such as Intel x86 (which has few registers to waste), store this pointer in a `thread_info` structure on the kernel stack (at the end of it) and, from it, can compute the address of the `task_struct`.

The `current` macro takes care of getting the current process in whatever way.

`preempt_count` is an important variable. We will see it when talking about kernel preemption.

`thread_struct` is used only during context switch to save callee saved registers and make appear the `switch_to` invocation like a normal function. However, it isn't normal since it will return not immediately but after the task is scheduled to run again.

Of course, also tasks have their states, and they are as follow:

- `TASK_RUNNING` – The process is runnable; it is either currently running or on a run-queue **waiting to run**.
- `TASK_INTERRUPTIBLE` – The process is sleeping (that is, it is blocked), waiting for some condition to exist. Wakes up and become runnable if it receives a signal.
- `TASK_UNINTERRUPTIBLE` – Does not wake up and become runnable if it receives a signal. Less often used than interruptible (only when event is expected to arrive rapidly). Note, tasks that are in this state are **not killable**.

- `TASK_DEAD` – Temporary state used at thread termination. When in this state, the thread will no longer be executed. Its metadata are kept in memory until its parent thread reaps it. This is necessary to pass the return value of a terminated thread. These are called **zombies**.

We mentioned a queue where tasks are waiting for being executed, it is called **wait queue**.
It is a structure that maintains tasks in a wait state for a specific event.

To change the state, they need a `wake_up` function that change the state, but we could have some problem like *Thundering Herd problem*. When a `wake_up` function is called, it wakes up all the tasks in the queue but, in some cases, only one task will be able to read that data; all the rest will simply wake up, see that no data are available, and go back to sleep.

For solving this problem, the kernel defines two different kinds of sleeping:

- **exclusive**, always put at the end of the queue. When wake up reaches them, it stops
- **non-exclusive**, are always woken up by the kernel

2.2 Tasks hierarchy

`fork()` creates a new copy of the current `task_struct`. The copy differs from the parent only in terms of:

- PID (which is unique) and PPID (parent's PID, which is set to the original process),
- certain resources, such as pending signals, which are not inherited.

Copy on write: rather than duplicate the process address space, the parent and the child can share a single copy; if data is written to, a duplicate is made, and each process receives a unique copy.

(threads are just processes that happen to not use copy on write)

The `init` task is responsible for starting up long running services, mount hard drives, perform clean-up a more. Several implementations exist but the most important to note are:

- SystemV
  - Set of processes to start are collected in run-levels. At any moment, the system is in one of them.
    - 1 (single user), 2 Multiuser (no net), 3 Multiuser, 5 X11, 6 Reboot
  - `/etc/inittab` describe which processes need to be on a run-level switch
  - The disadvantage is that the `init` task is single threaded, this means start-up times very long)
- SystemD
  - Compatible with SystemV but offers an alternative declarative environment and it is **parallelizable**
    - In *declarative environments* you specify what you want, and the system will figure out what it needs to do to get there.
    - In an *imperative environment* you specify all the steps you need to do to get there. It's the difference between writing a SQL statement and a for loop.

- systemd includes a tool called systemctl that allows you to query the status of services as well as start and stop them.

2.3 Task scheduling (basics)

A ***scheduling class*** is an API (set of functions) that include policy-specific code, e.g., functions to select the core on which the task must be enqueued (`select_task_rq`) and functions to actually put the task on that queue (`enqueue_task`).
This allows developers to implement thread schedulers without reimplementing generic code and also helps minimizing the number of bugs.

Which are these scheduling class?

- SCHED_DEADLINE
- SCHED_FIFO
- SCHED_RR
- SCHED_OTHER
- SCHED_BATCH
- SCHED_IDLE

If multiple policies have a runnable thread, a choice must be made by Linux to determine which policy has the highest priority. Linux chooses a simple fixed-priority list to determine this order (deadline → real-time → fair → idle).

The scheduler performs load balancing by migrating threads between cores in order to even the number of threads of all cores. Load balancing is done with a work stealing approach: each core does its own balancing and tries to steal threads from the busiest core on the system.

In 2007 has been introduced **Completely Fair Scheduler** for non-real-time processes. Still completely separated sets of processes with a certain normal priority π:

- **Real-time processes** $\pi \in [0, 99]$; they belong to scheduling class SCHED_FIFO and SCHED_RR
- **Non real-time processes** $100 \leq \pi(v) \leq 139$ which depend on a nice value $v \in [-20, +19]$:

$$\pi(v) = 120 + v$$

The central data structure of the core scheduler that is used to manage active processes is known as run-queue.
Multiple run-queues (one per CPU) are needed to avoid contention over task selection among processors. Practically, each class has a run-queue that contains runnable tasks. These are grouped in a general `struct rq` for each core.

The main scheduler function (`schedule()`) is invoked directly in many points in the kernel to allocate the CPU to a process other than the currently active one (e.g., after returning from system calls/interrupts or when a thread does some explicit blocking (mutex/semaphore/waitqueue)).

2.4 CFS (Completely Fair Scheduler)

Linux's CFS scheduler does not directly assign time-slices to processes. It assigns a proportion of the processor time depending on the load of the system.

How the proportion is assigned:

- this proportion is affected by each process's ***nice value v*** which acts as a weight, changing the proportion of the processor time each process receives
- when a process becomes runnable, if it has consumed a smaller proportion of the processor than the currently executing process, it runs immediately.

The idea is to share processor time through a weighted average which depends on the weight of the process and two parameters.

Targeted latency τ: is the overall time in which you would like that each of the processes has been given some time to work. Decreasing the targeted latency might increase responsiveness at the expense of context switching time.

Minimum granularity μ: a floor on the time-slice assigned to each process. By default it is 1 millisecond.

For each process *p*, its time-slice is computed as

$$\tau_p = f(\nu_0, \dots, \nu_p, \dots, \nu_{n-1}, \tau', \mu) \sim \max\left(\frac{\lambda_p \tau'}{\sum \lambda_i}\right)$$

where:

- $\lambda_i(\nu_i)$ is the weight associated with a process, a sort of priority.
- τ_p is given by the `sched_slice` function.

Accounting update: on each timer interrupt at time t, CFS updates the variable `vruntime` (ρ_i) and `sum_exec_time` (ϵ_i) of the current process with the time elapsed since the last measurement $\Delta t = (t - t_i)$.

ρ is a **measure of the dynamic priority of the process** and depends on the time it has consumed but also its weight. Higher-weighted task (identified by a higher λ_i dividend) are put nearer the leftmost side but only if they don't starve lower priority ones. Why using ρ? All processes have consumed their share τ_p only if they have reached the same amount of ρ.

Unblocked processes are assigned the minimum value of all ρ's in the rb tree. In practice, it is easy to see if a process got less allocated time, even with different weights, because it will have a lower ρ.

Accounting effect: when $\epsilon_i = \tau_i$ or the process blocks or a new process with lower ρ becomes ready, the next process is selected to run; typically, this is the one with the smallest ρ taken from a red-black tree (a container of tasks sorted by ρ). Insertion in the tree is always $O(\log n)$ while the minimum run-time is always cached in a variable.

But we have a problem, CFS is not enough to guarantee optimal CPU usage, for example, assume

- user A with 2 threads and user B with 98 threads

User A will be given only 2% of the time (not good!). Each user should get an equal share of the CPU, and this share is then divided among the user's threads. This is also true for all other resources in the systems.

The solution is **Cgroups**.
Cgroup stands for control group and is a group of tasks. Linux allow to create, assign and throttle resources assigned to each control group and it can be useful when we have classes of tasks for which we know we must enforce limits of some kind.

When we have to assign processes, we have also the problem of **load balancing** in order that all CPUs do equal amount of work. In fact, we

- can't balance on the same number of threads
- can't balance on λ_q of each run queue q.

The scheduler performs load balancing by migrating threads between cores in order to even the number of threads of all cores. Load balancing is done with a work stealing approach: each core which becomes idle does its own balancing and tries to steal threads from the busiest core on the system. In this context, this core is called the designated core.

Load balancing will start from a designated core which will try to pull-in tasks from other crowded cores to even out the average load. Each core becomes designated:

- periodically (through a `SCHED_SOFTIRQ` set up by `scheduler_tick`) or
- when it becomes idle

It does this in two steps, `find_busiest_group()` to find the busiest group (frequently the busiest subdomain) and then `find_busiest_runqueue()` to find the actual run-queue in the group from which to steal the task.

2.5 Other OS processes

In FreeRTOS, a task is just a function that does not return. It can be in 4 states:

1) Ready
2) Suspended (any task may put any task in this mode)
3) Running
4) Blocked (is entered on events being waited on)

The scheduling works with a *fixed priority preemptive* with time slicing for equal priority processes but preemption can be disabled.

3. Multi-process programming and Inter-Process Communication (IPC)

3.1 Recap

- Program → A sequence of computer instruction stored somewhere (not in execution)
- Application → User-oriented concept of computer program. It is usually used for programs with a GUI
- Process → An instance of a program, that is currently in execution.
    - It has an isolate memory address space
    - It contains one or more threads
- Thread → The smallest schedulable unit of execution
    - Belong to the memory address space of a process
    - It shares the memory address space with other threads of the same process
        - can be synchronized
        - can access global memory areas
        - can access variables of other threads
        - they have also private address space, but they can communicate only with the help of the OS (IPC)
    - It runs sequentially on the CPU (but there are exceptions like hw parallelism)
- Task → Task has no a unique definition. It usually means a single unit of computation at conceptual level. In Linux is a synonymous of thread

3.2 Forking and executing processes

The forking is the operation whereby a process spawns a new process which is a copy of itself. So,

- A new process is created and runs concurrently
- The virtual address space is copied
    - All the variables have the same value with the exception of the `fork()`'s return value
- Most of the physical pages in the memory are marked as "copy-on-write"
- No way to directly access parent ↔ child variables (we need IPC)

Every process has exactly one parent and may have an arbitrary number of children, the only exception is the `init` process which is the ancestor of all processes.

Every process has:

- **PID**: Process IDentifier
- **PPID**: Parent Process Identifier

In Linux, the PID is of `pid_type` defined in `<sys/types.h>` and it is a 32-bit integer and limited by `/proc/sys/kernel/pid_max` special file.

But how our machine boots?

1) The system is powered on, the firmware (BIOS/EFI) is loaded into main memory

2) The BIOS or EFI firmware performs all the necessary checks and launches the bootloader
3) The bootloader mounts the necessary filesystem and loads the correct kernel image (usually `/boot/vmlinuz-*`)
4) The kernel starts running

So, in Linux, the last step is when the `init` process is started. It is the first user-space process launched at the end of the kernel boot.

- It has always PID = 1 and PPID = 0
- It is the ancestor of all user-space processes in the system
- It starts all enabled services during startup
- Most common implementations are:
  - SysVinit (legacy)
    - Load all services from `/etc/init.d` and `/etc/rc.d/`
  - System
    - Load services according to `/etc/systemd/`

So, when we want to execute a program:

1) init forks itself
2) load the `exec*()` family of functions (**execve** syscall)
3) all the function in the family take the executable path as first argument
4) then execute the commands inside the program

3.3 Parent-Child basic synchronization

The simplest one for parent-child relation is the `wait()` based mechanism.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Two functions perform simple parent-child synchronization:

- `wait()` - Suspend the execution until one of the children terminates
- `waitpid()` - Suspend the execution until a specific child process terminates (or changes state)
- `status`: pointer to a variable where to write the return value; it can be NULL
- `pid`: PID of the child to wait
- `options`: extra options

In this case we could have *zombie processes*, in fact when a process terminates, the Linux kernel changes the state to "zombie" and saves its return value. We have 2 cases:

1) It returns to the parent the child return value when the parent calls `wait()` (or `waitpid()`) on the child process
2) If the parent terminates before calling `wait()`, the child is adopted by init

a. The init process performs `wait()` on all children freeing the memory and the PID number

Persistent zombie processes in the system are a sign of a programming error.

3.4 Inter-Process Communication

In Linux, there are two libraries:

- POSIX (newer)
- System V (legacy)

Signals

They are:

- Unidirectional
- No data transfer
- The information content is only the "signal type"
- Asynchronous

Signals are communication methods between processes, and they can be sent by processes or by the OS.

POSIX signals	Number	Default action	Description
`SIGHUP`	1	Terminate	Controlling terminal disconnected
`SIGINT`	2	Terminate	Terminal interrupt
`SIGILL`	4	Terminate and dump	Attempt to execute illegal instruction
`SIGABRT`	6	Terminate	Process abort signal
`SIGKILL`	9	Terminate	Kill the process
`SIGSEGV`	11	Terminate and dump	Invalid memory reference
`SIGSYS`	12	Terminate and dump	Invalid system call
`SIGPIPE`	13	Terminate	Write on a pipe with no one to read it
`SIGTERM`	15	Terminate	Process terminated
`SIGUSR1`	16	Terminate	User-defined signal 1
`SIGUSR2`	17	Terminate	User-defined signal 2
`SIGCHLD`	18	Ignore	Child process terminated, stopped or continued
`SIGSTOP`	23	Suspend	Process stopped

Examples:

- A terminating child process sends a `SIGCHLD` to the parent
- A user pressing Ctrl+C on the keyboard sends a `SIGINT` to the process

To send a signal we use the `kill()` function with return value of 0 on success and -1 on error

```
#include <signal.h>
#include <sys/types.h>
int kill(pid_t pid, int sig);
```

- `pid`: PID of the receiver process
- `sig`: signal to send (a SIG_* constant)

To handling a signal we use the `sigaction()` with same return value as before

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

- `signum`: signal to catch (a SIG_* constant)
- `act`: new settings to apply to register a handler function
- `oldact`: output variable, it saves the old settings if not NULL

We have also **signal handler registration**

```
struct sigaction{
                  void (*sa_handler)(int);
                  void (*sa_sigaction)(int, siginfo_t *, void *);
                  sigset_t sa_mask;
                  int sa_flags;
                  void (*sa_restorer)(void);
                 };
```

- `sa_handler`: handler function (or `SIG_IGN`)
- `sa_sigaction`: alternative handler. It allows you to specify a handler that accept an input data. Flags must contain `SA_SIGINFO`
- `sa_mask`: set a mask to block certain signals
- `sa_flags`: various options
- `sa_restorer`: not intended to be used by user applications, not in POSIX, do not use

There is also an extension thanks to modern Linux systems that support POSIX real-time. It introduces these functions:

- `sigqueue()` → send a queued signal
- `sigwaitinfo()` → synchronously wait a signal
- `sigtimedwait()` → synchronously wait a signal (for a given time)

We can also mask signals, in fact they can be masked (blocked) to avoid disrupting your code execution. This is similar to `SIG_IGN`, but instead of begin dropped, they are enqueued and managed later when the process unmasks the signal but `SIGKILL` and `SIGSTOP` cannot be masked.

```
int sigprocmask(int how, const struct sigset_t *set,
                struct sigset_t *oldset);
```

- `how`: `SIG_BLOCK` (add to the mask), `SIG_UNBLOCK` (remove a signal from the mask), `SIG_SETMASK` (replace the mask)
- `set`: set of signals

- `oldset`: output value, previous set of signals

(Unnamed) Pipes and (named pipes) FIFO

Based on the producer/consumer pattern, one producer writes and one consumer reads, so it is unidirectional. Data are written/read in a First-In-First-Out (FIFO) fashion.

(In Linux, the OS guarantees that only one process at a time can access the pipe)

How to create a PIPE

```
#include <unistd.h>
#include <fcntl.h>
int pipe(int pipefd[2]);
int pipe2(int pipefd[2], int flags);
```

- `pipefd`: array of 2 integers to be filled with two file descriptors (FD)
  - `pipefd[0]`: file descriptor of the **read end** of the pipe
  - `pipefd[1]`: file descriptor of the **write end** of the pipe
- `flags`:
  - `O_CLOEXEC`: close file descriptor if exec(…) is called
  - `O_DIRECT`: perform I/O in "packet mode"
  - `O_NONBLOCK`: avoid blocking read/write in case of empty/full pipe

How to use a PIPE directly with low-level I/O functions

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
ssize_t read(int fildes, void *buf, size_t nbyte);
```

Or you can transform your FD to a stream

```
#include <stdio.h>
FILE *fdopen(int fildes, const char *mode);
```

We can use all the f* functions like fwrite, fscanf, …

FIFO pipes have the same behaviour of unnamed pipes but they are based on special files created in the filesystem and not on file descriptors and the data are transferred like reading/writing to a disk file, but no actual I/O is performed: the OS passes data from one process to another.

How to create a FIFO

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

- `pathname`: the path + filename of the file to create

- `mode`: the file permissions to give to the file
    - `S_I*` constants help you (e.g., `S_IWUSR` – write permission for the owner 0200)

Just use open/write/read to access the FIFO.

Message queues

In this type of queues, we have multiple writers and multiple readers of a queue based on a priority. The status of the message can be observable, and all the special file of the queues are in the system-wide directory `/dev/mqueue/`.

How to create/open a Message queue

```
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);
```

- `name`: a unique name for the message queue (starting with /)
- `oflag`: opening flag (`O_RDONLY`, `O_WRONLY`, `O_CREAT`, …)
- `mode`: the file permissions to give to the file (only for `O_CREAT`)
- `attr`: attributes
- It returns a message queue descriptor (mqd_t is an opaque data type) or -1 in case of error

Message Queue Attributes

```
struct mq_attr {
   long mq_flags; // 0 or NON_BLOCK
   long mq_maxmsg; // max nr. messages in the queue
   long mq_msgsize; // max message size in bytes
   long mq_curmsgs; // nr. messages currently in the queue
};
```

Other management functions

```
#include <mqueue.h>
int mq_close(mqd_t mqdes);
int mq_unlink(const char* name);
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
               struct mq_attr *oldattr);
```

How to send a message

```
#include <mqueue.h>
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
unsigned int msg_prio);
```

- `mqdes`: message queue descriptor
- `msg_ptr`: pointer to message to send

- `msg_len`: the length of the message (in bytes)
- `msg_prio`: non-negative priority value in the range [0; 31]

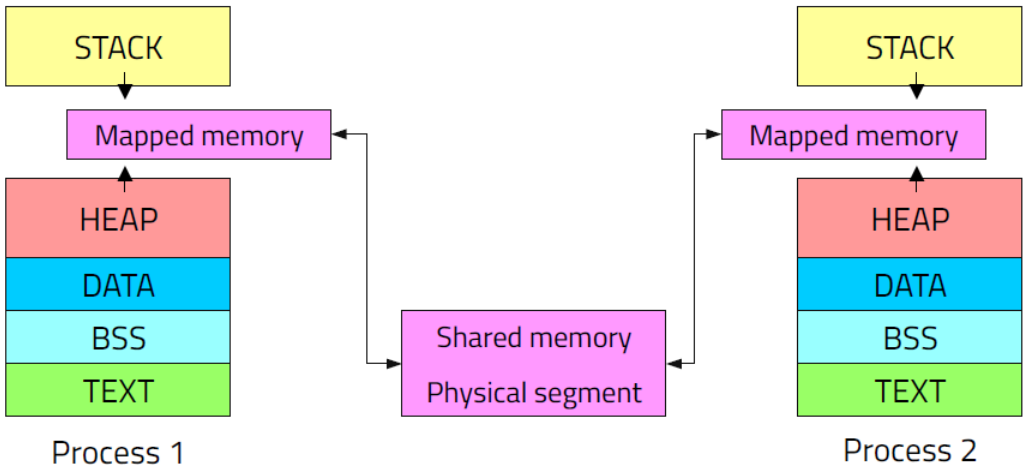
How to receive a message

```
#include <mqueue.h>
int mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
               unsigned int *msg_prio);
```

- `mqdes`: message queue descriptor
- `msg_ptr`: output parameter – pointer to a buffer to fill
- `msg_len`: the length of the buffer (in bytes)
- `msg_prio`: output parameter – the priority of the message

Shared memory

Shared memory is an IPC mechanism that allows two processes to share a memory segment.
In POSIX the shared memory is based on the memory mapping concept.



Like message queues, opening/creation of shared memory segments are referenced by name and, in Linux, a special file is created under `/dev/shm/<name>`.

How to create/open a shared memory

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
int shm_open(const char *name, int oflag, mode_t mode);
```

- `name`: a unique name for the shared memory (starting with /)
- `oflag`: opening flag (`O_RDONLY`, `O_WRONLY`, `O_CREAT`, …)
- `mode`: the file permissions to give to the file
- The function returns a file descriptor

After the creation of a shared memory object, we need to specify the size of the special file

```
#include <unistd.h>
#include <sys/types.h>
int ftruncate(int fd, off_t length);
```

- `fd`: the file descriptor to truncate
- `length`: size in bytes

Mapping the memory

```
#include <sys/mman.h>
void * mmap(void *addr, size_t length, int prot, int flags,
            int fd, off_t offset);
```

- `addr`: starting virtual address or NULL (usually want it to be NULL and leave the kernel to select it)
- `length`: size of the mapped segment
- `prot`: memory protection flag → `PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`
- `fd`: file descriptor
- `offset`: offset to skip from the begin of the file descriptor
- `flags`: visibility of the updates w.r.t. other processes
  - `MAP_SHARED`
  - `MAP_PRIVATE`

How to clean up the memory

```
#include <sys/mman.h>
int munmap(void *addr, size_t length);
int shm_unlink(const char *name);
```

- `munmap()`: it deletes the mappings for the specified address range
- `shm_unlink()`: it removes the shared memory object created by `shm_open()`

Synchronization

How to synchronize two processes?

The `wait()`/`waitpid()` based approach is clearly very limited. Synchronization is needed to avoid race conditions on shared resources.

POSIX provides inter-process semaphores

- semaphore counter = 0 → WAIT
- semaphore counter > 0 → PROCEED

When counter can be 0 or 1, the semaphore is called *binary semaphore* and behaves similarly to a mutex.
We have two *atomic* functions:

- `wait()` → block until counter > 0, then decrement it and proceed
- `post()` → increment the counter

How to initialize and destroy an unnamed semaphore

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);

int sem_destroy(sem_t *sem);
```

- `sem`: output parameter – semaphore data structure to initialize
- `pshared`: if 0 shared among threads, if not 0 shared among processes
- `value`: initial value
- It returns 0 on success, -1 on error

How to initialize and destroy a named semaphore

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflags);
sem_t *sem_open(const char *name, int oflags, mode_t mode,
                unsigned int value);

int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

- `name`: the POSIX object name
- `oflags`: opening flags (`O_CREAT`, `O_EXCL`)
- `mode`: file permissions
- `value`: initial value
- It returns the pointer to the semaphore object or `SEM_FAILED` in case of error
- `sem_close()`: closes the named semaphore
- `sem_unlink()`: remove the named semaphore object

How to synchronization semaphores

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *timeout);
```

- `sem`: the semaphore object
- `timeout`: time limit to wait if semaphore counter == 0
- `sem_trywait()`: is the non-blocking version of `sem_wait()`
- All functions return 0 for success or -1 in case of error

4. Task scheduling (Part 1)

4.1 Introduction

Preemption

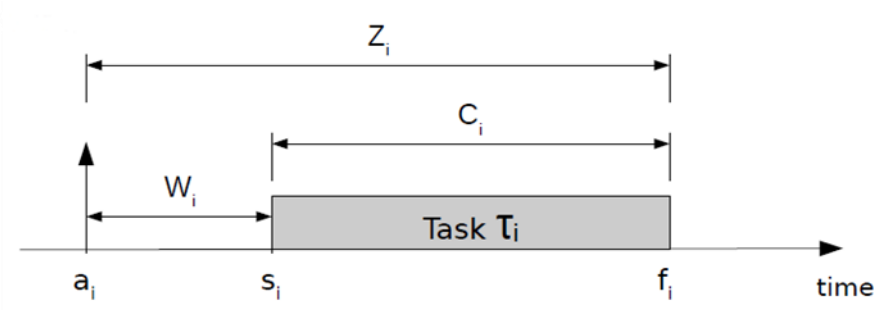
- Operation for temporarily suspending the execution of a task in order to execute another task
- Can be task-triggered (unusual) or OS-triggered (more common)
- Supported in most of the modern OS

I/O blocking

- The task can voluntarily suspend their execution because waiting for I/O data (e.g., keyboard input, file input/output, etc.)
- The CPU is available to run other tasks

4.2 Task model

We have a set of tasks $T = \{\tau_1, \tau_2, \dots, \tau_i, \dots, \tau_n\}$ and some parameters as shown



- a_i Arrival time → Time instant at which task is ready for execution and put into the ready queue
- s_i Start time → Time instant at which execution actually starts
- W_i Waiting time → Time spent waiting in the ready queue
- f_i Finishing time → Time instant at which the execution terminates
- C_i Computation time → Amount of time necessary for the processor to execute the task without interruptions
- Z_i Turnaround time → Difference between finishing time and arrival time
  (Z_i is not necessarily $W_i + C_i$; in case of preemption or suspension Zi contains also the interferences from the other tasks and the time the task is interrupted e.g., the I/O waiting time)

Depending on the type of operations dominating the lifetime of task, we may identify the "boundness" of a task:

- CPU-bound → the task spends most of its time executing operations
- I/O bound → the task spends most of its time waiting for I/O operations

A computing system is composed of:

- m processing elements (PE) $CPU = \{CPU_1, CPU_2, \dots, CPU_m\}$
  - each PE, at each time t, is assigned to zero or one task
- s additional resources: $R = \{R_1, R_2, \dots, R_s\}$
  - each resource, at each time t, is assigned to zero or more task

The scheduler aims at optimizing one (or more) objectives, for this reason we have some metrics:

- Processor Utilization: percentage of time the CPU is busy
- Throughput: number of tasks completing their execution per time unit
- Waiting time (avg): Average time the tasks spent in the ready queue
- Fairness: Do the tasks have a fair allocation of the processor?
- Overhead: Amount of time spent in taking scheduling decisions and context-switches

One problem to avoid is *starvation*. It is the undesirable perpetuated condition in which one (or more) tasks cannot execute due to the lack of resources.

4.3 Classification of scheduling algorithms

PREEMPTIVE	NON-PREEMPTIVE
• Running tasks can be interrupted (by the OS) at any time to allocate the process to another active task • Necessary if we need a responsive system	• Once started, a task is executed until its completion • Scheduling decisions can be taken only when tasks terminate • It guarantees minimum overhead • Definitely not good for a responsive system

STATIC	DYNAMIC
• Scheduling decisions are based on fixed parameters, whose values are known before task activation • Strong assumptions required	• Scheduling decisions are based on parameters that typically changes at run-time, during task execution • May need a run-time feedback mechanism

OFFLINE	ONLINE
• The scheduler is executed on a set of known tasks before their activation • The output is the sequence of tasks to execute, called schedule	• The scheduler is executed at run-time • It can schedule new previously unknown tasks

• It must be static (we need to know all the parameters) • Very limited, but often necessary if you want to provide some formal guarantees	

OPTIMAL	HEURISTIC
• The scheduler is based on an algorithm optimizing a given cost function, defined over the task set • The algorithm may be too complex → high overhead	• Algorithms are based on a heuristic function • Tending to optimal scheduling, but without any guarantee about achieving it • Generally, much faster than optimal algorithms

4.4 Scheduling algorithms

First-In-First-Out (FIFO)

How it works

- Tasks are scheduled in the order of arrival
- Non-preemptive
- Also known as First Come First Served (FCFS)

Advantages

- Very simple
- It does not require any knowledge of the processes

Disadvantages

- Not good for responsiveness
  - Long tasks may monopolize the processor
  - Short tasks are penalized (and may even starve)

Shortest Job First (SJF)

How it works

- Tasks are scheduled in ascending order of computation time (Ci)
- Non-preemptive
- Also known as Shortest Job Next (SJN)

Advantages

- Optimal non-preemptive w.r.t. the minimization of the average waiting time

Disadvantages

- Risk of starvation for long tasks
- We need to know C_i in advance

Bringing forward any task with lower execution time makes the average waiting time smaller. Performing this operation for all the possible task, we get the SJF schedule, which is, consequently, the optimal scheduler for minimizing the waiting time.

Shortest Remaining Time First (SRTF)

How it works

- Preemptive variant of SJF
- It uses the remaining execution time instead of C_i to decide which task to dispatch

Advantages

- Improve responsiveness for all tasks compared to SJF

Disadvantages

- Risk of starvation for long tasks
- We need to know C_i in advance

Highest Response Ratio Next (HRRN)

How it works

- Select the task with the highest Response Ratio:
  - $RR_i = (W_i + C_i)/C_i$
  - Non-preemptive

Advantages

- Prevent starvation with respect to SJF

Disadvantages

- We need to know C_i in advance

Round Robin (RR)

How it works

- Tasks are scheduled for a given *time quantum q* (also called time slice). When the time quantum expires, the task is preempted and moved back to the ready queue.
- Preemptive

Advantages

- Computable maximum waiting time: $(n-1) * q$
- No need to know C_i in advance
- Good to achieve the fairness and responsiveness goals
- No starvation is possible

Disadvantages

- Turnaround time worse than SJF

Quantum value choice

- Long quantum
  - Tend to FIFO scheduler
  - Favours CPU-bound tasks
  - Low overhead (less context switches)
- Short quantum
  - Reduce average waiting time
  - Favours I/O-bound tasks
  - Good for responsiveness and fair scheduling
  - High overhead (more context switches)

5. Task scheduling (Part 2)

5.1 Priority-based algorithm

Priority is a task parameter through which we can specify the importance of a task (let us name it Pi)

- fixed (known at design-time), or
- dynamic (changes at run-time)

The priority is usually expressed with an integer value:

- the lower the integer value, the higher the priority
- the higher the integer value, the lower the priority

Multi-level Queue Scheduling

- For each queue we can specify a different scheduling algorithm (e.g., RR or FCFS)
- The first task to schedule is picked from the topmost non-empty queue (highest priority)
- Tasks cannot be moved from one ready queue to another
- The queue scheduling is preemptive
- Risk of starvation!
    - While highest priority queues are populated by new tasks, the scheduling of tasks in lower priority queues is delayed

Multi-level Queue Scheduling: Round Robin

Priority is selected depending on the workload type:

- CPU-bound tasks have low priority (therefore high quantum value)
- I/O-bound tasks have high priority (therefore low quantum value)

This priority scheme guarantees the responsiveness (but starvation is always a risk)

How to determine if a task is CPU-bound or I/O-bound?

- Information provided by the user
- Information provided by the program itself (but it requires a run-time feedback mechanism)

Multi-level Feedback Queue Scheduling: Dynamic priority

The scheduling works like the previous multi-level RR scheme. The priority now is dynamic, changing according to this rationale:

- The new/activated the task is moved to the highest priority queue (lowest quantum value)
- If the quantum of the running task expires, the task is moved to the next queue with lower priority (higher quantum value)

CPU-bound tasks are progressively moved in queues with longer time quantum.

But we have not solved the problem of starvation, how to solve it in multi-level scheduling?

Multi-level Feedback Queue Scheduling: Time slicing

Each queue gets a maximum percentage of the available CPU time it can use to schedule the task, which determines a time quota. If the time quota expires, the remaining tasks in the queue are skipped, and we start picking tasks from the next (lower priority) queue.

- $\sum quota_i$ can be larger than the period but
  - it is guaranteed to have no starvation* if $\sum quota_i \leq period$
  - However, we can still have starvation due to the scheduling policy of one of the queues

Multi-level Queue Scheduling: The case of Linux scheduler

Limited with a time slice fashion:

- `quota = /proc/sys/kernel/sched_rt_runtime_us`
- `period = /proc/sys/kernel/sched_rt_period_us`

Scheduling policy is set by the user/system administrator. If you choose IDLE for your task, you are aware that the task may starve.

Multi-level Feedback Queue Scheduling: Aging

The priority of the task is increased as long as it spends time in the ready queue (it gets older…). Prevent a task from being indefinitely postponed by new coming higher priority tasks (it avoid starvation).

5.2 Multi-Processor Scheduling

The scheduler must choose the task to execute, and the processor to assign. This kind of decision is very hard because new problems occur like:

- task synchronization may occur across parallel executions
- difficult to achieve high level of utilization of the whole set of processors (or CPU cores)
- simultaneous (not only concurrent!) access to shared resources
  - often hierarchical and multi-level (e.g., *cache memories*)

[Cache memory is a type of high-speed memory that is used to store frequently accessed data. It allows the processors to quickly access the data they need, which can improve the overall performance of the system.

When a processor needs to access data, it first checks the cache memory to see if the data is already stored there. If it is, the processor can access the data directly from the cache, which is much faster than accessing it from main memory. This can help the processor to execute its instructions more quickly, improving the performance of the system.

If the data is not stored in the cache, the processor will have to access it from main memory. This is a slower process, but it ensures that the data is still available to the processor when it needs it.

The use of cache memory in a multi-processor system can help to improve the overall performance of the system by allowing processors to access the data they need more quickly.]

Single queues vs Multiple queues

Single queue

- All the ready tasks wait in the same global queue

Advantages

- Simple design
- Good for fairness
- Good for managing CPU utilization

Disadvantages

- Scalability
    - Scheduler runs in any processor, and it requires a synchronized version of the ready queue (a proper semaphore/mutex is required)

Multiple queue

- A ready queue for each processor
- More scalable approach
- Potentially more overhead (more data structures to handle)
- Easier to exploit data locality (processor affinity)
- Possibility of adopting a single global scheduler or per-CPU schedulers
- Need of *load balancing*

Load balancing

Unbalanced ready queues have negative impact for several reasons

- CPU utilization: processors may be idle (due to empty queue), while other queues have waiting tasks
- Performance: waiting times and response times can be reduced by moving the tasks in a different queue (e.g., with a lower number of tasks)
- Thermal management: balancing the ready queue levels the temperature distribution
- Thermal management has an impact on power consumption, energy efficiency and reliability

Load balancing is typically performed via **task migration**:

- tasks are moved to a different queue (usually containing less tasks)
- as we have already seen, this has a consequence on the cache overhead and interference

There two possible implementations of task migration:

1) Push model: a dedicated task periodically checks the queues' lengths and moves tasks if balancing required
2) Pull model: each processor notifies an empty queue condition and picks tasks from other queues

a. *Work stealing* is an example of pull model-based approach. Each per-CPU scheduler can “steal” a task from the other in case of empty queue
b. Scalable in theory but we need to protect the concurrent access to the ready queues with locks. How to determine the queue from which to take a task?

Hierarchical queues

A global queue dispatching tasks in local ready queues → Hierarchy of schedulers

- Better control over CPU utilization / load balancing
- Good scalability
- More complex to implement

6. General concurrency

6.1 Taxonomy

You have *concurrency* when your program is composed by activities where one activity can start before the previous one has finished.
Different parts or units of a program can be executed in an overlapped way without affecting the final outcome but ensuring to have different program counters, stacks and registers for variables.

One way to overlap these activities is through the **threading model**. It is not the only one enabling concurrency. In fact it might incur context switch costs and might be too tight with respect to application requirements.

One motivation for supporting concurrency via the thread execution model is overcoming single-thread challenge. But they can be heavy because they involve the OS.

If you need something more lightweight, you can use continuations and callbacks. They are expressible in terms of "*suspended computation*" and "*continuation*".

6.2 Concurrency issues and prevention – Introduction

We characterize a program with two properties (which might be true or false):

- Safety (correctness), nothing bad happens. If your program is a state machine, we don't reach an error state or work on invalid data.
- Liveness (progress), eventually something good happens. In an FSM, we eventually reach a final state.

An issue that happens is *data race*. You have a data race whenever you have two instructions a and b (in different threads) where at least one of them is a write and when there is nothing that enforces either $a < b$ or $b < a$.

6.3 Concurrency issue and prevention – Deadlock

It happens when no task can act because it is waiting for another task to take action.

We have a deadlock when we reach all the following conditions:

1) Mutual exclusion → two threads can't act on the same resource at the same time.
2) Hold-and-wait → threads hold resources allocated while waiting for additional resources.
3) No preemption → resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
4) Circular wait → there exists a circular chain of waiting threads (like dining philosophers' problem).

How to prevent them (respectively)?

1) In some cases, it is possible to design various data structures without locks at all.
2) Through appropriate APIs, a task can voluntarily release the resource if the acquisition of further resources fails (is valid also for "3) No preemption").

4) This deadlock can be solved making the last philosopher invert its order, in fact, we to guarantee that no cyclical wait arises.

In general, there are several ways to avoid or resolve deadlocks:

- Prevention (use resource allocation policies that ensure that a process can only request resources that it can safely acquire)
- Detection and recovery (detect and resolve the deadlock. This might involve aborting one or more processes)
- Avoidance (use resource allocation policies that ensure that a deadlock cannot occur in the first place)
- Prevention and detection

6.4 Concurrency issue and prevention – Priority inversion

It is a scenario in scheduling in which a high priority task is indirectly superseded by a lower priority task effectively inverting the assigned priorities of the tasks. This violates the priority model that high-priority tasks can only be prevented from running by higher-priority tasks. Inversion occurs when there is a resource contention with a low-priority task that is then preempted by a medium-priority task.

There three techniques to solve this issue:

1) Highest Locker Priority (HLP)
    a. When a task holds a critical resource, its priority is changed to the ceiling priority value of the critical resource. If a task holds multiple critical resources, then maximum of all ceiling priorities values is assigned as priority of the task.
2) Priority Inheritance (PIP)
    a. when a task goes through priority inversion, the priority of the lower priority task which has the critical resource is increased by the priority inheritance mechanism. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion.
3) Priority Ceiling (PCP)
    a. A task T_i is allowed to enter a critical section only if its priority is higher than all the priority ceilings of the semaphores currently locked by other tasks.
    A priority ceiling of a semaphore is the highest priority among the taskt that can lock it.

7. Linux user space concurrency

7.1 Sleeping locks implementation – Futex

A *futex* (short for "*fast userspace mutex*") is a kernel system call that programmers can use to implement basic locking, or as a building block for higher-level locking abstractions such as semaphores and POSIX mutexes or condition variables.

The goal is to:

- avoid system calls, if possible, since system calls typically consume several hundred instructions
- avoid unnecessary context switches
- avoid thundering herd problems (all threads waiting are woken up)

The state of the lock (i.e., acquired or not acquired) can be represented as an atomically accessed flag in shared memory. In the **uncontended case**, a thread can access or modify the lock state with atomic instructions, for example atomically changing it from not acquired to acquired using an atomic compare-and-exchange instruction.
In the other hand (**contended case**), a thread may be unable to acquire a lock because it is already acquired by another thread. It then may pass the lock's flag as a futex word and the value representing the acquired state as the expected value to a `futex()` wait operation. This `futex()` operation will block if and only if the lock is still acquired (i.e., the value in the futex word still matches the "acquired state"). When releasing the lock, a thread has to first reset the lock state to not acquired and then execute a futex operation that wakes threads blocked on the lock flag used as a futex word.

7.2 Event-based concurrency

This is a different style of concurrent programming. The idea is:

- wait for something to occur
- check what type of event arrived
- do the small amount of work it requires
- rinse and repeat

The event loop is a programming construct or design pattern that waits for and dispatches events or messages in a program. It is useful because there is no ocking, no context switch, no uncontrollable preemption and, when an event is processed, only one activity taking place in the system.

To check whether there is any incoming network I/O, it uses either `select` or `poll` APIs.

Blocking network IO

- No call that blocks the execution of the caller can ever be made in `processFD`;
- Only one event is being handled at a time, there is no need to acquire or release locks

```
void main()
{
   while(1)
   {
      int wfd[] = {10, 20, 30};
      fd_set readFDs;
      FD_ZERO(&readFDs)
      for(int i = 0; i < 3; i++)
         FD_SET(wfd[i], &readFDs);
      int rc = select(3, &readFDs, NULL, NULL, NULL);
      for (int i = 0; i < 3; i++)
      {
         if (FD_ISSET(wfd[i], &readFDs)) processFD(wfd[i]);
      }
   }
}
```

Non-blocking network IO

If an event handler issues a call that blocks, the entire server will do just that: block until the call completes. When the event loop blocks, the system sits idle, and thus is a huge potential waste of resources.

How to solve this?
The idea is to run either run blocking file I/O operations in a thread pool and make the processFD register a callback for when data is ready.
Use Linux/Posix AIO. Provides native functions for launching asynchronous File I/O operations and be notified in several ways. (Difficult to write cross-platform apps)

8. Kernel space concurrency

Interrupts

An interrupt is an event that alters the normal execution flow of a program and can be generated by hardware devices or even by the CPU itself. When an interrupt occurs the current flow of execution is suspended, and interrupt handler runs. After the interrupt handler runs the previous execution flow is resumed.

Two categories:

- synchronous, generated by executing an instruction (called named exceptions)
- asynchronous, generated by an external event (named interrupts)

If the interrupt and the interrupted task are using the same resource, then we must regulate access.

Multiprocessing

Multiprocessing support implies that kernel code must simultaneously run on two or more processors.

Code in the kernel, running on two different processors, can simultaneously access shared data at the same time. This time it is called *true concurrency*.

Code that is safe from concurrency on symmetrical multiprocessing machines is SMP-safe.

8.1 Kernel preemption

- In a **non-preemptive**, one kernel code runs until completion.
- In a **preemptive** kernel, a process running in kernel mode might be replaced by another process.

The preemption points are:

- at the end of interrupt/exception handling, when `TIF_NEED_RESCHED` flag in the thread descriptor has been set (forced process switch)
- if a task in the kernel explicitly blocks

How do we ensure to force process switch only when it safe?
Preemption is managed through the `preempt_count` in the `thread_info` struct of the current process; this is checked at preemption points. A non-zero counter tells to the kernel that it cannot perform a context switch.
This variable gets incremented:

- every time a thread acquires a lock in kernel mode
- whenever the kernel is beginning the execution of an interrupt. Only when the last level of interrupt is returning then preemption can be enabled

Real-time patches use `PREEMPT_RT`. The key point of the `PREEMPT-RT` patch is *to minimize the amount of kernel code that is non preemptible*, while also minimizing the amount of code that

must be changed in order to provide this added preemptibility.
In particular, critical sections and interrupt handlers become normally preemptible.

- *Spinlocks*: complete kernel preemption (Reduces scheduling latency by replacing most of the spinlocks with blocking mutexes). In fact, with `PREEMPT_RT`, `spinlock_t` and `rwlock_t` will become sleeping locks
- *Interrupts*: with mainline Linux, when an interrupt occurs, CPU is preempted and ISR is executed. Current interrupt line is masked off while the ISR pushes time consuming activities to kernel threads.

8.2 Locking (single processor)

Spinlocks

A spinlock is a lock that causes a thread trying to acquire it to simply wait in a loop ("*spin*") while repeatedly checking whether the lock is available. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting. Once acquired, spinlocks will usually be held until they are explicitly released, although in some implementations they may be automatically released if the thread being waited on (the one that holds the lock) blocks or "*goes to sleep*".

Because they avoid overhead from operating system process rescheduling or context switching, spinlocks are efficient if threads are likely to be blocked for only short periods. For this reason, operating-system kernels often use spinlocks.

Reader-writer spinlocks

If your data accesses have a very natural pattern where you usually tend to mostly read from the shared variables, the reader-writer locks (`rw_lock`) versions of the spinlocks are sometimes useful. They allow multiple readers to be in the same critical region at once, but if somebody wants to change the variables it has to get an exclusive write lock.

Seqlocks

A seqlock (short for sequence lock) is a special locking mechanism used in Linux for supporting fast writes of shared variables between two parallel operating system routines.

It is a *reader–writer* consistent mechanism which avoids the problem of writer starvation.
A seqlock consists of storage for saving a sequence number in addition to a lock. The lock is to support synchronization between two writers and the counter is for indicating consistency in readers. In addition to updating the shared data, the writer increments the sequence number, both after acquiring the lock and before releasing the lock.

Sleeping locks

Semaphores in Linux are sleeping locks. When a task attempts to acquire a semaphore that is unavailable, the semaphore places the task onto a wait queue and puts the task to sleep.
When the processor is free to execute other code; semaphores must be obtained only in process context because interrupt context is not schedulable.

8.3 Lockless

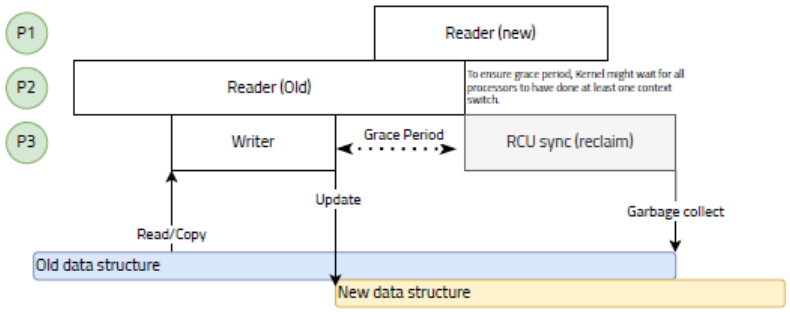
The goal is to reach low latency value of reads to shared data that is frequently read and non-frequently written.

The idea is:

- for readers NO locks
- for writers create a new version of data structure and publish it with a single atomic instruction

Linux uses a "*Read, Copy, Update*" (RCU) mechanism to implement lockless.
The basic idea behind RCU (read-copy update) is to split a destructive operation into two parts:

1) removes references to data items within a data structure and can run concurrently with readers (**removal**). This takes advantage of the fact that writes to single aligned pointers are atomic and must consider memory ordering and must consider memory ordering
2) carries out the destruction (**reclamation**).

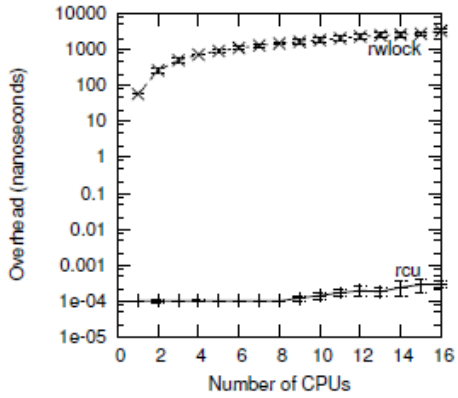


A "*grace period*" must elapse between the two parts, and this grace period must be long enough that any readers accessing the item being deleted have since dropped their references.

The RCU update sequence goes something like the following:

a. Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it.
b. Wait for all previous readers to complete their RCU read-side critical sections.
c. At this point, there cannot be any readers who hold references to the data structure, so it now may safely be reclaimed

Comparison with readwrite locks

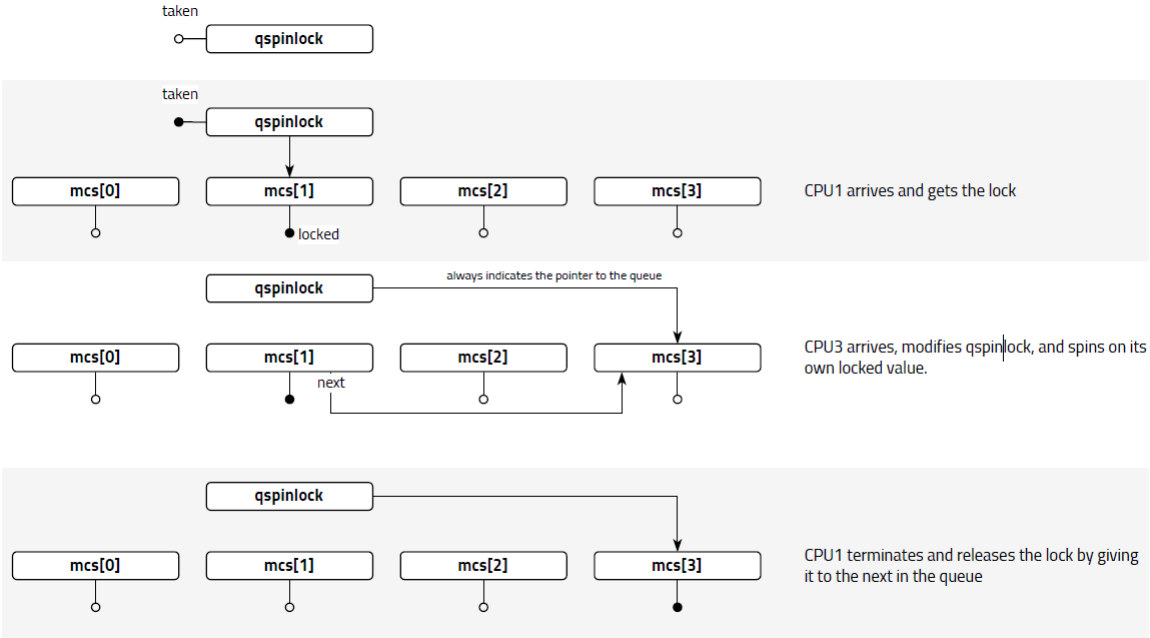


The figure shows the overhead of entering and completing an RCU critical section, and acquiring and releasing a readwrite lock.

PRO → avoid deadlocks and makes read lock acquisition a breeze

CON → might delay the completion of destructive operations

8.4 Multiprocessors locking



A Mellor-Crummey and Scott (MCS) lock is a type of lock that is used to implement mutual exclusion in a multi-threaded program. The goals of an MCS lock are to provide a high-performance synchronization mechanism for multi-threaded programs and to minimize contention between threads.

The structure of an MCS lock consists of a queue of threads waiting to acquire the lock, with each thread represented by a node in the queue. When a thread attempts to acquire the lock, it adds itself to the queue and spins (repeatedly checks the lock status) until it becomes the head of the queue. Once a thread becomes the head of the queue, it can acquire the lock and proceed with its critical section of code.

They are generally considered to be more efficient than other types of locks, such as spin locks and mutexes, because they allow threads to block rather than spin when they are unable to acquire the lock. This can help to reduce the overall contention and overhead of the lock, leading to improved performance in high-concurrency environments.

8.5 Memory models

A memory model defines the behaviour of the visibility (and consistency) of the operations done by one thread from another thread.

Due to write buffering, speculation and cache coherency protocols of modern processors, the order in which memory accesses are seen by another thread occur might be different from the order in the issuing thread.

Sequential consistency

Define $<_p$ the program order of the instructions in a single thread and $<_m$ as the order in which these are visible in the shared memory (either order is also referred to as the "*happens-before*" relation).

A multiprocessor is called **sequentially consistent** if and only if for all pairs of instructions (Ip,i, Ip,j) you have that

$$I_{p,i} <_p I_{p,j} \rightarrow I_{p,i} <_m I_{p,j}$$

In practice the operations of each individual processor appear in this sequence in the order specified by its program.

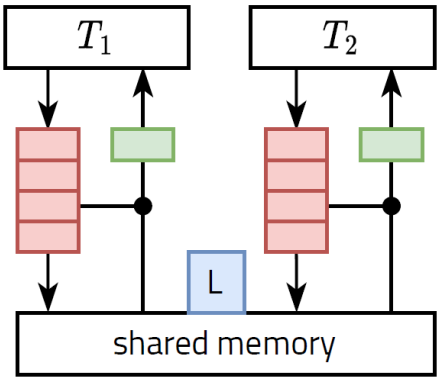
Total Store Order

TSO is the model of the INTEL X86 memory consistency.

- Uses local write queue to hide memory latency
- A memory read consults the local write queue
    - but cannot see the write queues on other processors
    - a processor sees its own writes before others do.

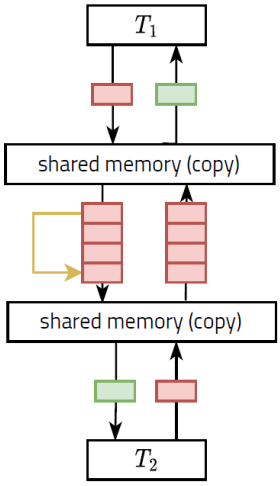
All processors do agree on the (total) order in which writes (stores) reach the shared memory and when a write reaches shared memory, any future read on any processor will see it and use that value.
Write order is preserved by the write queue, and because other processors see the writes to shared memory immediately.

On a TSO, it can happen that th1 and th2 queue their writes and then read old values from memory before either write makes it to memory, so that both reads see zeros.
Some synchronization algorithms not based on locking (Dekker's algorithm or Peterson's algorithm) may fail if one thread isn't seeing all the writes from another.



Partial Store Order

It is the ARM memory model and it is even weaker than TSO.
Each processor reads from and writes to its own complete copy of memory and each write propagates to the other processors independently, with reordering allowed as the writes propagate. This means that th1's writes may not be observed by other threads in the same order.



(the yellow arrow means that a store can bypass another store)

8.6 Data races

In TSO and PSO one can use synchronization instructions to avoid wrong reads. To do use we can use ***fences*** (or barriers) to enforce program order into memory order. Of course putting fences everywhere is not a good idea because this can lowering the performance.
A better thing is to identify when putting a fence is actually needed.

Hardware memory model

A hardware memory model defines the behaviour and the visibility (and consistency) of the operations of a machine language program done by one thread from another thread. No compiler is involved.

You have a data race whenever you have two instructions *a* and *b* (in different threads) where at least one of them is a write and when there is nothing that enforces either $a <_m b$ *or* $b <_m a$.

So, we want to avoid this problem and we need some synchronization operation in order to enforce memory order. We have two types of these operations:

1) *Release operation*: a sync operation that, when observed, makes the observer conclude that all previous writes have been completed. This is generally a store that ensures that write happens after any previously executed reads or writes.
2) *Acquire operation*: a sync operation that must be used by the observer to recognize the release operation. This is a load that ensures that the read happens before any subsequent reads or writes

Software memory model

Compilers can introduce additional reordering of instructions that might appear as if the machine had a weaker memory model. This adds another layer of complexity to hardware memory models.

Higher level languages must give to the programmer a way to enforce ordering happens-before relations just as it is done at the ISA level. This is called the ***language memory model***.

A software memory model defines the behaviour and the visibility (and consistency) of the operations of a high-level language program (and an appropriate run-time) done by one thread from another thread.
For example, a compiler might reorder writes to unrelated variables. This is perfectly fine for single thread applications but might make a multi-threaded program behave like it was running on a PSO machine.

9. Linux Memory Mechanics

9.1 Virtual address space

Each process has its own page directory with both user-mode and kernel mappings. In user mode, kernel pages cannot be accessed while, on context switch, only user space mapping changes.

- **Kernel logical addresses** are mapped directly to physical addresses starting from 0. These correspond to contiguous physical memory and can be used by DMA if needed. There is a special function called `kmalloc` requesting pages from here.
- **Kernel virtual addresses** are not contiguous in physical memory. They are used to allocate large buffers where finding chunks of physical memory would be difficult. The `vmalloc` function is used to request non-contiguous physical pages and sets up addresses here

Kernel allows the use of multiple page sizes, not just the standard 4KB page: 2MB, 1GB (aka *huge pages*).

On 32bit machines the amount of physical memory that can be mapped above user space is less than 1GB; this is called *low memory*. To access the other memory, you must map it into the virtual memory space.

In theory, a page table is all the kernel needs to implement virtual memory. However, page tables represent only pages that are present in memory, not all the pages of a process. So, how can I understand if a VPN is valid but not mapped? Or, how do I get the corresponding data?
Linux uses ***Virtual Memory Areas*** (VMA).

A VMA can be classified as:

- Mapped to file (with backing store) or not mapped (anonymous, i.e., stack, heap)
- Shared or private.
- Readable, writable or exec. For example, the object code for a process might be mapped with `VM_READ` and `VM_EXEC` but not `VM_WRITE`.
- `VM_IO` specifies if that the area is a mapping of a device's I/O space. This field is typically set by device drivers when `mmap()` is called on their I/O space.

(inode 0 pages correspond to sections that do not have a backing store)

Backing store areas:

- Come from the PT_LOAD segments in the ELF file.
- Initially code, read only data and initialized writable data.
- Writable initialized data is put read only. COW is then used if written (This happens also for the child of a forked process)

Anonymous areas (heap, stack and bss) are originally mapped into a zero page maintained by the OS and mapped as read only.

- A `VMA_GROWSDOWN` area is initialized with a certain number of zero pages and it can grow when the last page is accessed. It is used for the stack.

- For other Anon VMAs, use explicitly brk or sbrk.

We have *page fault* when a process accesses a page does not present in memory.

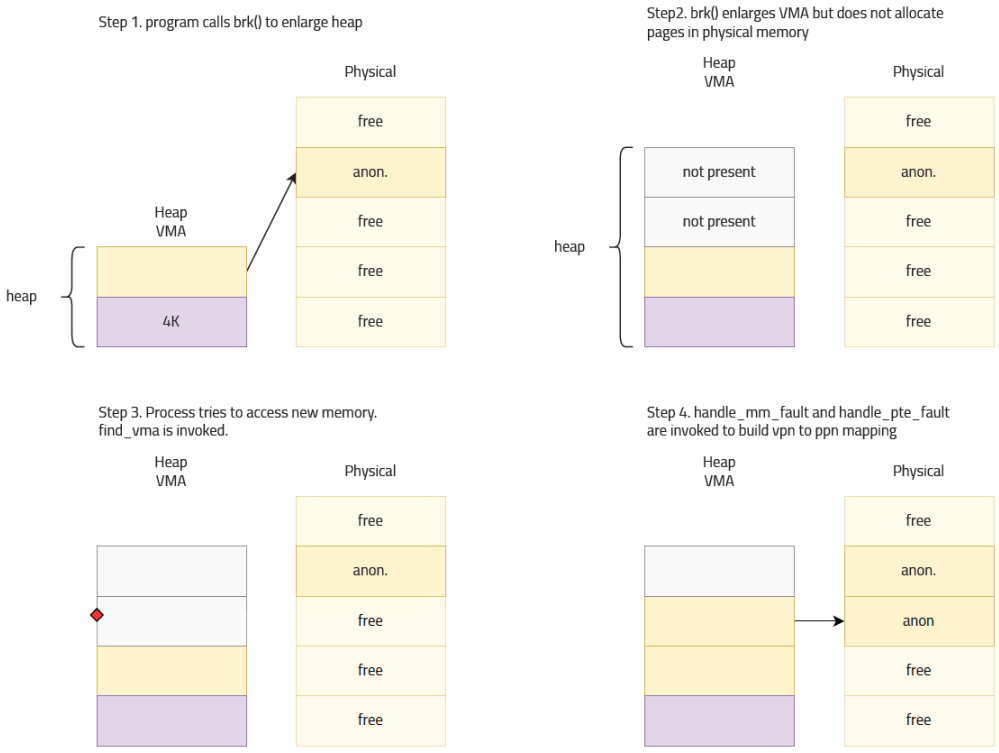
How to find a vma?
It uses a red-balck tree of vmas to dinf the corresponding VMA. Why red-black tree? Because they are easily balanced.

On success, permissions are checked since a user cannot access memory assigned to kernel, for example.

How to create VMAs?
VMAs can be created explicitly by a process by calling `mmap()` on an already opened file descriptor.

```
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

`mmap` allows to avoid the kernel copying data into user space. In fact, we are given direct access to its internal page cache.
The function returns a pointer to the beginning of a region of virtual memory where the contents of the file are located. The OS will try to find space for the page and make it accessible by updating the page table of the process accordingly (demand paging below).



9.2 Physical address space

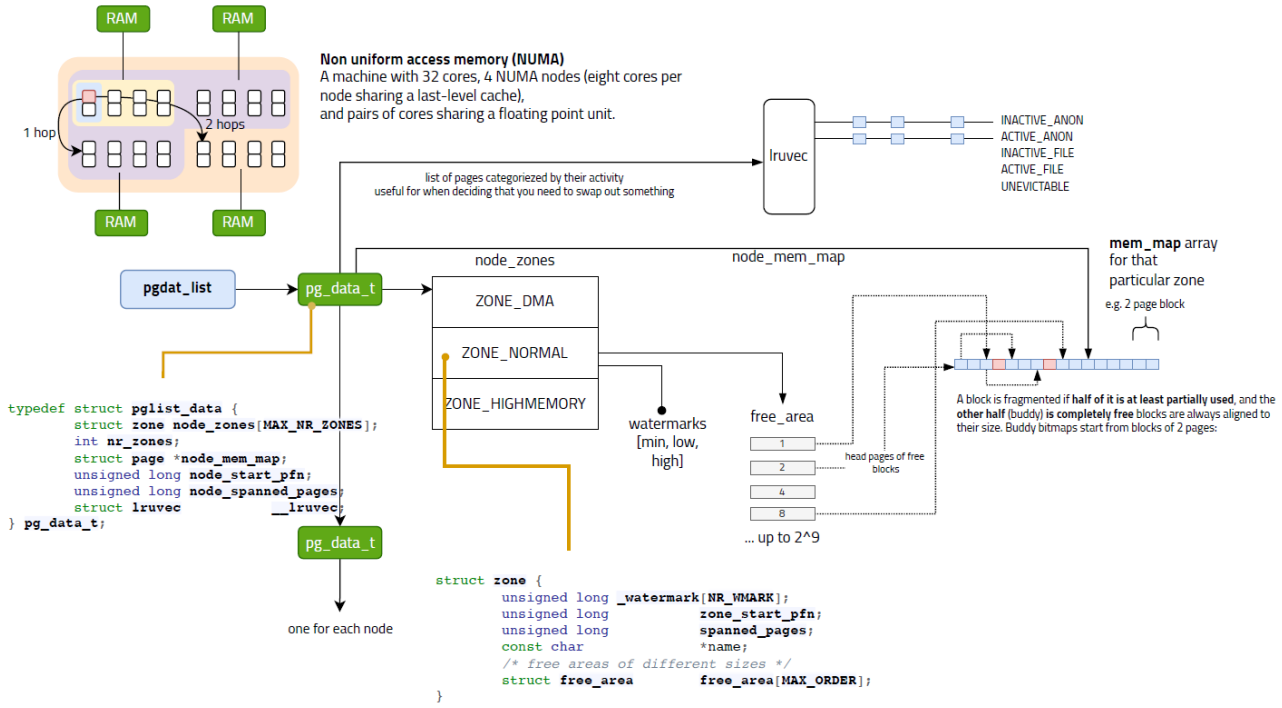
Memory may be arranged into ***NUMA*** banks or nodes. All physical pages in the system are represented by a struct page in an array `mem_map` (which is split in different portions, one for each node, on NUMA system).

Of course, accessing in memory has a cost and we have different costs depending on the "*distance*" from the processor. Linux uses a *node-local allocation* policy to allocate memory from the node closest to the running CPU. As processes tend to run on the same CPU, it is likely the memory from the current node will be used.

There are different zones:

- `ZONE_DMA` is memory in the lower physical memory ranges which certain ISA devices require.
- `ZONE_NORMAL` is directly mapped by the kernel into the upper region of the linear address space
- `ZONE_HIGHMEM` is the remaining available memory in the system and is not directly mapped by the kernel.

Zone information are contained in a list called `pgdat_list` and for UMA architectures like PC desktops, only one static `pg_data_t` structure called `contig_page_data` is used.



9.3 Page allocation

Each zone contains the total size of pages in the zone and an array of lists of free page ranges. When kernel needs contiguous pages and there are free pages, the ***buddy algorithm*** is used to choose how to split free memory regions to satisfy the request.

Buddy algorithm

It tries avoiding as much as possible the need to split up a large free block to satisfy a request for a smaller one for two reason, one, in the kernel we might need contiguous pages for DMA and, second, we increase the chance of using 4MB pages can be used instead of smaller ones, reducing TLB misses.
In this way it is easy to allow to merge back blocks when they are freed.

How the allocation and deallocation work?
To allocate a block of a given order we check the free list of the specified order and all higher orders.

- If a block is found at the specified order, it is allocated immediately.
- If a block of higher order must be used, then we divide the larger block into two 2 (order-1) blocks, add the lower half to the appropriate free list, and allocate the memory from the upper half, executing this step recursively if necessary.

When freeing memory, we check whether the block being freed has a free buddy block; if so, we combine the two blocks into a single free block by removing the adjacent block from its free list and then freeing the larger block; again, this process is performed recursively if necessary.

Linux uses:

- buddy bitmaps to represent the fragmentation status of the next level of block size
- free lists for each block order to access directly blocks of free pages

Bitmaps make it quick and easy to check whether a block being freed has an unused buddy or not.

Page cache

Assume you have multiple processes accessing the same file.
How does Linux know if a data requested by one process is already in memory? It maintains a ***per-file page cache***.

It is the set of physical pages that are the result of reads and writes of regular filesystem files stored in a structure called the `address_space`. (each `address_space` has a unique radix tree stored as `page_tree`.)

$$file\ descriptor + offset \rightarrow physical\ page\ mapping$$

How to reclaim a page?
An algorithm called *Page Frame Reclaim Algorithm* is used.
It tries to relieve memory pressure caused by file-backed pages in physical memory that changes dynamically. Keep two lists of clean pages for each zone, active and inactive. Victims are taken from the inactive list and must be not dirty, pages go into the active list only after two accesses. Linux periodically moves the pages from active to inactive trying to keep lists balanced.

9.4 Object allocation

In general, within the kernel, fixed size data structures are very often allocated and released.

The Buddy System that we presented earlier clearly does not scale:

- internal fragmentation can rise too much (one page for a single object is too much)
- the buddy system on each NUMA node is protected by a (spin)lock

There are two fast allocators in the kernel:

1) Quicklists, used only for paging
2) Slab Allocator, used for other buffers

Quicklists

Quicklists are used for implementing the page table cache, i.e., pages that we need for the data of the page table itself. For the three functions `pgd/pmd/pte_alloc()` we have three quicklists `pgd/pmd/pte_quicklist` per CPU. Each architecture implements its own version of quicklists, but the principle is the same.

Quicklists use a LIFO (Last-In First-Out) approach. During the allocation, one page is popped off the list, and during free, one is placed as the new head of the list. This is done while keeping a count of how many pages are used in the cache.

Slab allocator

The general idea behind the SLAB allocator is to have caches of commonly used objects kept in an initialized state available for use by the kernel.
The SLAB allocator consists of a variable number of caches, linked together by a doubly linked list called cache chain.
Every cache manages objects of particular kind. Each cache maintains a block of contiguous pages in memory called *Slab*.

The slab allocator provides two main classes of caches:

- **dedicated**: these are caches that are created in the kernel for commonly used objects (e.g., `mm_struct`, `vm_area_struct`, etc…). Structures allocated in this cache are initialised and when they are freed, they remain initialised so that the next allocation will be faster.
- **generic** (size-N and size-N(DMA)): these are general purpose caches, which in most cases are of sizes corresponding to powers of two. This separation can be seen in the dedicated file for slab in the proc file system.

10. Linux Memory Security

10.1 Buffer overflows

The success of these attacks lays in part on efficacy of the attacker to know where other parts of the system code located.

```
int some_function(char *input)
{
   char dest_buffer[100];
   strcpy(dest_buffer, input); // oops, unbounded copy!
}
```

In this kind of attacks the problem is that, these programs, can force the execution of arbitrary binaries by manipulating the stack. For example, writing `dest_buffer` can be used to rewrite the return address of `some_function`.

One of the countermeasures is ASLR that changes the position of the stack randomically.

This hold also for the kernel, in fact, we have KASLR but there three problems with it:

1) Kernel ASLR (KASLR) currently only randomizes where the kernel code is placed at boot time.
2) The amount of randomization that can be applied to the base address of the kernel image is rather small due to address space size and memory management hardware constraints.
3) The sheer amount of other information leaking bugs present in all kernels and the almost complete lack of prevention mechanisms against such leaks. I.e., you can probably lookup the current address in some way.

10.2 Meltdown

The kernel memory can contain sensitive information from another application, like a password. In principle, kernel memory is ***all*** the memory.

Meltdown demonstrates that processor speculation can leak kernel memory into user mode long enough for it to be captured by a side-channel cache attack.

It relies on a CPU race condition that can arise between instruction execution and privilege checking. Put briefly, the instruction execution leaves side effects that constitute information not hidden to the process by the privilege check. The process carrying out Meltdown then uses these side effects to infer the values of memory mapped data, bypassing the privilege check.

One countermeasure is **KPTI**; it uses two separate user-space and kernel-space entirely.

- One PGD includes both kernel-space and user-space addresses same as before, but it is only used when the system is running in kernel mode. Protection against execution (SMEP) and access invalid references (SMAP) is in place.

- The second PGD, *for use in user mode*, contains a copy of user-space and a minimal set of kernel-space mappings that provides the information needed to enter or exit system calls, interrupts and exceptions.

11. Virtualization

11.1 Introduction

A Virtual Machine is an efficient, isolated duplicate of the real machine dedicated to an OS. Based on a virtual machine monitor (VMM) that creates an environment for an OS that is essentially identical with the original machine with only minor speed decrease, and it has a complete control of system resources.

Key aspects:

- Fidelity → same behaviour of the real machine
- Safety → the virtual machine cannot override the VMM's control of virtualized resources
- Efficiency → programs should "show at worst only minor decreases in speed"

Why a virtual machine?

Because

- Consolidate and partition hardware
    - Use one physical machine at 100% instead of two at 50% (consolidation)
- React to variable workloads
    - Reduced hardware and administration costs for datacentres
    - Horizontal scalability
- Standardized infrastructure
- Security sandboxing and fault tolerance

Host system: the OS where virtual machines run
Guest system: the OS that runs on top of the virtual machine
Virtual machine monitor (VMM) or Hypervisor: software program that translates/mediates access to physical resources such as interrupts or sensitive processor state. Ensures isolation.
Type 1 Hypervisor: also called native hypervisor; runs on bare metal without any OS abstraction
Type 2 Hypervisor: runs in the context of another OS (think about KVM or VirtualBox)

Instruction types: unprivileged and privileged. The latter are those that trap in user mode
Virtualization idea: run privileged instructions in a de-privileged mode.
An instruction is **virtualization-sensitive** if it is:

- control sensitive: it directly modifies the machine status (e.g., enabling or disabling interrupts, modifying the interrupt vector table)
- behaviour sensitive: instructions that behave differently when used in either user or supervisor mode. It might affect fidelity.

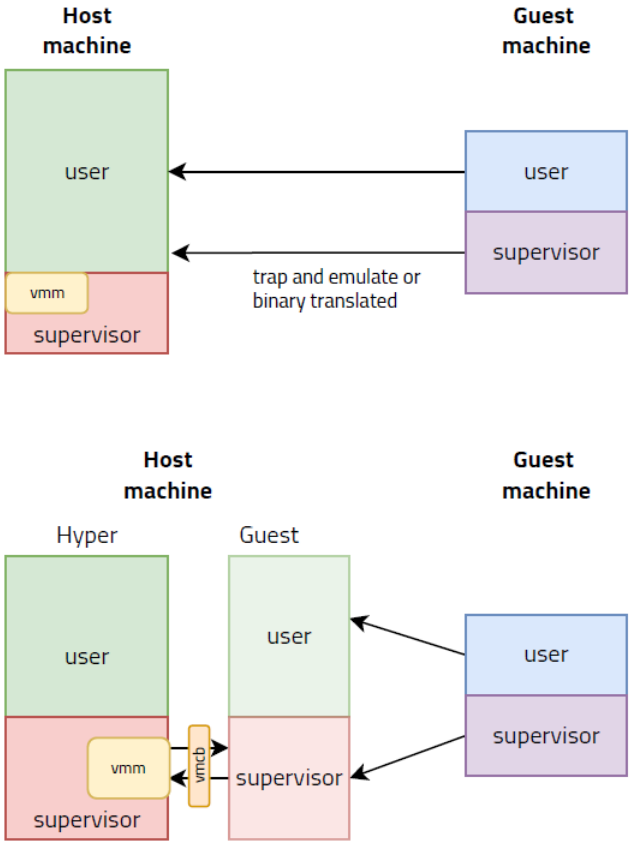
We can have two kinds of virtualizations:

1) **software-based**: the guest user code runs on the processor
    a. Guest privileged instructions must be intercepted and emulated (trap-and-emulate) by the hypervisor. This is the classical definition of full system virtualization (1974).

A classical VMM executes guest operating systems directly, but at a reduced privilege level.

b. Binary translation if trap and emulate is not possible, i.e., virtualization sensitive instructions that are not privileged.

2) **hardware-assisted**: certain processors provide hardware assistance for CPU virtualization.
   a. Hardware-assisted virtualization reduces hypervisor intervention to the minimum possible.
   b. Processors provide two additional guest/hypervisor modes which are orthogonal to user/supervisor modes.
   c. Guest/supervisor mode runs almost at native speeds and only on certain events it enters hypervisor/supervisor mode (typically on page table manipulation).



1 The upper is sw based, the lower is hw based

11.2 Software based virtualization

Software-based virtualization means deprivileging and shadowing:

- `g.supervisor` is translated into `h.user`
- privileged instruction or memory access produces an interceptable trap
- `h.supervisor` installs its own structures (shadow) instead of those dictated by `g.supervisor`

The guest operating system cannot access data structures of the host operating system.

A number of key data structures used by a processor need to be **shadowed**. Shadow data structures are used mainly for virtualization.

What if Guest OS modifies its page table? Should not allow it to happen directly.

- VMM needs to intercept when guest OS modifies page table, and update the shadow page table accordingly
- Mark the guest table pages as read-only (in the shadow page table) and manage corresponding traps. This technique is called *Memory Tracing*.

Speeding it up

Some architectures have virtualization-sensitive unprivileged instruction, e.g., Intel x86.

On Intel x86 we have 4 kinds of unprivileged virtualization-sensitive instructions:

1) `pushf`, `popf`, `iret`: Instructions manipulating the interrupt flag (%eflags.if) are NOPs if executed in user mode. They do not trap!
2) `lar`, `verr`, `verw`, `lsl`: provide visibility into segment descriptors in the global or local descriptor table
3) `pop <seg>`, `push <seg>`, `mov <seg>`: manipulate segment registers. This is problematic since the privilege level of the processor is visible in the code segment register
4) `sgdt`, `sldt`, `sidt`, `smsw`: provide read-only access to privileged registers such as %idtr.

There are some problems with pure trap and emulate.

Ring aliasing: a guest OS could easily determine that it is not running at supervisor privilege level.
Address space compression: OSs expect to have access to the processor's full virtual address space however VMM must have a minimal number of pages allocated in the guest address space to manage traps.
Excessive faulting: on x86-32, `sysenter` and `sysexit` are used for each system call but trap into the VMM any time they are executed by the guest OS.

A solution is a binary translator that converts an input binary instruction sequence into a second binary instruction sequence that can execute natively on the target system. A dynamic binary translator performs the translation at run-time by storing the target sequences into a buffer called the translation cache.
Binary translation is used to rewrite in terms of ring 1 instructions certain ring 0 instructions.

11.3 Hardware assisted virtualization

The main goals are:

- avoid the problems of deprivileging
- allow the state of the guest can be explicitly and comprehensively saved and resumed and which is used for some shadow structure
- improve performance

The advantages of this kind of virtualization are:

- When you have many system calls, HW prevails because they run without VMM intervention
- Recovering the guest state to manage traps is easier with support from hardware
- No binary translation means less overhead and less memory

The drawbacks are:

- Some instruction needs to trap, be fetched and executed by the VMM while in software they can be directly transformed into an emulation routine (e.g., memory tracing)
- IO requires a full vmm/guest round trip while in software they can be made operate on a virtual chipset
- The control path on page faults (which is similar for both HW and SW) could be higher overhead for hardware than software

Speeding it up

Virtual machines often make use of direct device access ("device assignment") when configured for the highest possible I/O performance. From a device and host perspective, this simply turns the VM into a user space driver, with the benefits of significantly reduced latency, higher bandwidth, and direct use of bare-metal device drivers 3.

The VFIO driver exposes direct device access to user space, in a secure, IOMMU protected environment. In other words, this allows safe non-privileged, user space drivers. Many modern systems now provide DMA and interrupt remapping facilities to help ensure I/O devices behave within the boundaries they've been allotted. This includes x86 hardware with AMD-Vi and Intel VT-d, POWER systems with Partitionable Endpoints (PEs) and embedded PowerPC systems such as Freescale PAMU.

KVM takes away the problem of creating your own vmm. It consists of a loadable kernel module, `kvm.ko`, that provides the core virtualization infrastructure and a processor specific module, `kvm-intel.ko` or `kvm-amd.ko`. Originally a forked version of QEMU was provided to launch guests and deal with hardware emulation that isn't handled by the kernel. That support was eventually merged into the upstream project.

Real-world examples

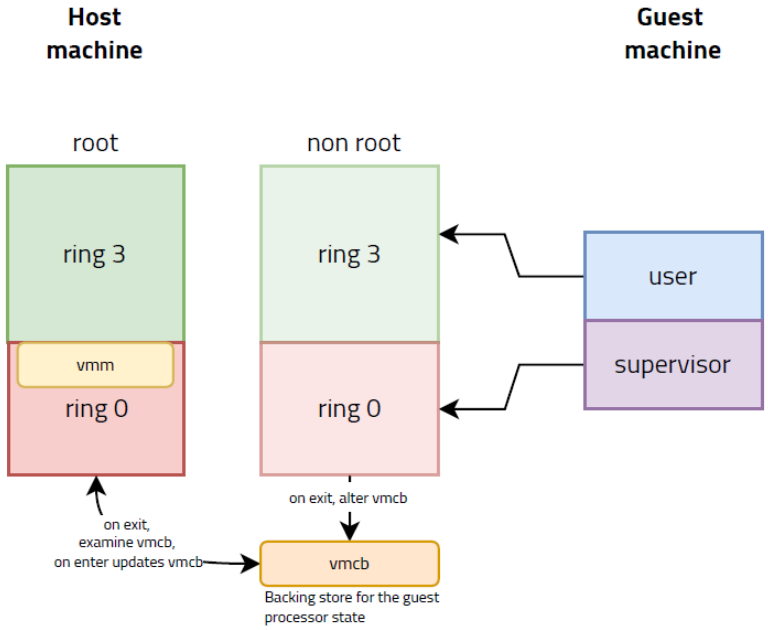
Intel Vanderpool Technology, referred to as VT-x, represents Intel's virtualization technology on the x86 platform. Its goal is a simplified VMM software by closing virtualization holes by design.

- ring compression (lack of OS/Applications separations if only 2 rings are used)
- non-trapping instructions (some instructions at ring 1 are not trapped, for example popf)
- excessive trapping

Sensitive instructions do trap but where possible, privileged instructions affect state within the virtual CPU as represented within the VMCB, rather than unconditionally trapping.
In the Linux kernel the module that is in charge of enabling the support for VT-x (or AMD-V) is KVM, that makes the kernel able to work as an hypervisor.

The virtual machine control block (VMCB) contains:

- **Guest state** (state of the processor saved and restored exiting and entering the VM)
- **Host state** (state of the processor to be restored when exiting the VM)
- **Execution control**:
    - Specifies what should happen if an interrupt arrives when the guest OS is running; do we let it manage it?
    - Specifies whether some instructions (e.g., manipulating cr3) are to be interpreted as sensitive
- **Exit reason**: why the VM exited (e.g., on I/O access which register was involved etc.)
- **Enter and exit control**: used, for example, if the machine in root mode received an interrupt but the guest has disabled interrupt, tell the VMM that it must trap when it reenables it



11.4 Beyond Virtualization – Paravirtualization

Paravirtualization is a virtualization technique that presents a software interface to the virtual machines which is similar, yet not identical, to the underlying hardware–software interface.

The intent of the modified interface is to reduce the portion of the guest's execution time spent performing operations which are substantially more difficult to run in a virtual environment compared to a non-virtualized environment.
The paravirtualization provides specially defined *hooks* to allow the guest(s) and host to request and acknowledge these tasks, which would otherwise be executed in the virtual domain (where execution performance is worse). A successful paravirtualized platform may allow the virtual machine monitor (VMM) to be simpler (by relocating execution of critical tasks from the virtual domain to the host domain), and/or reduce the overall performance degradation of machine execution inside the virtual guest.

11.5 Containerization

Containers are a way to isolate a set of processes and make them think that they are the only ones running on the machine.

The machine they see may feature only a subset of the resources actually available on the entire machine (e.g., less memory, less disk space, less CPUs, less network bandwidth).

Containers **are not virtual machines**:

- Processes running inside a container are normal processes running on the host kernel.
- There is no guest kernel running inside the container
- You cannot run an arbitrary operating system in a container, since the kernel is shared with the host (Linux, in our case).

The most important advantage of containers with respect to virtual machines is performance: there is no performance penalty in running an application inside a container compared to running it on the host.

Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.
The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources. Resources may exist in multiple spaces. Examples of such resources are process IDs, hostnames, user IDs, file names, and some names associated with network access, and interprocess communication.

Namespaces are a fundamental aspect of containers on Linux. In this way each process in Linux has its own network namespace, pid namespace, user namespace and a few others.

12. System architecture and IO

12.1 Introduction

A bus is a hardware connection channel. Originally, CPU was connected to a single highspeed bus. Other buses were bridged-on.

Bus categories:

- Memory bus (only connects the memory)
- General, high speed, I/O bus (e.g., Peripheral component interface)
- A peripheral bus, such as SCSI, SATA, or USB connects slow devices to the system, including disks, mice, and keyboards.

The faster a bus is, the shorter it must be. Components that demand high performance (such as the graphics card) are nearer the CPU.

Nowadays, point to point connections with CPU are preferred instead of a shared bus.

12.2 CPU to device communication

We use simple digital I/O ports (such as the standard PC parallel port) to show how the I/O instructions work and normal frame-buffer video memory to show memory-mapped I/O.

We chose simple digital I/O because it is the easiest form of an input/output port. Also, the parallel port implements raw I/O and is available in most computers: data bits written to the device appear on the output pins, and voltage levels on the input pins are directly accessible by the processor.

Of course, each device is assigned a `port` number in the I/O address space which names the device.

We have a problem because I/O instructions are limited, and this kind of instructions are slow.

12.3 Device to CPU communication

Polling

It indicates the cyclical verification of all input/output devices or devices by the operating system of a personal computer by testing the bus bits associated with each device, followed by any interaction (write or read).

This activity takes up a lot of the processor (CPU) operating time, which slows down the entire system.

Interrupt

It is a request for the processor to interrupt currently executing code (when permitted), so that the event can be processed in a timely manner. If the request is accepted, the processor will suspend its current activities, save its state, and execute a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is often temporary,

allowing the software to resume normal activities after the interrupt handler finishes, although the interrupt could instead indicate a fatal error.

Interrupts are commonly used by hardware devices to indicate electronic or physical state changes that require time-sensitive attention. Interrupts are also commonly used to implement computer multitasking, especially in real-time computing. Systems that use interrupts in these ways are said to be interrupt-driven.

When there are too many interrupts, the OS might *livelock*, that is, find itself only processing interrupts and never allowing a user-level process to run and actually service the requests.

When using programmed I/O to transfer a large chunk of data to a device, the CPU is overburdened with a rather trivial task and might waste time. Let's introduce an additional device that can orchestrate transfers between devices and memory without CPU intervention (Direct Memory Access - DMA). OS would program the DMA engine by telling it where the data lives in memory, how much data to copy, and which device to send it to.
(PCI allows any peripheral to have a DMA)

13. Linux interrupts

First, there is a difference between interrupts and exceptions. The first ones are issued by interval timers and I/O devices, instead, the other ones are caused either by programming errors by anomalous conditions that must be handled by the kernel.

Then we have two types of interrupts:

- *Asynchronous* (I/O) interrupts are generated by other hardware devices at arbitrary times with respect to the CPU clock signals
    - *Maskable interrupts* → all Interrupt Requests (IRQs) issued by I/O. A masked interrupt is ignored by the control unit as long as it remains masked
    - *Non-maskable interrupts* → always recognized by the CPU
- *Synchronous* interrupts are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction

Interrupt flow

When a device or event generates an interrupt, the interrupt is sent to the CPU, which stops its current task and executes a special routine known as an *interrupt handler*. The interrupt handler is responsible for determining the source of the interrupt and taking appropriate action, such as servicing a device or handling a software event.

On a multi-core system, interrupts are typically handled by a dedicated interrupt controller, which directs the interrupt to the appropriate CPU core. The interrupt controller uses a priority scheme to determine which interrupts should be handled first, with higher priority interrupts taking precedence over lower priority interrupts.

Once the interrupt has been handled, the CPU returns to its previous task. If the task was preempted by the interrupt, it will resume where it left off. If the task has completed, the CPU will move on to the next task in the queue.

In summary, the interrupt flow on Linux involves the following steps:

1) A device or event generates an interrupt
2) The interrupt is sent to the CPU
3) The CPU stops its current task and executes an interrupt handler
4) The interrupt handler determines the source of the interrupt and takes appropriate action
5) The CPU returns to its previous task

Programmable Interrupt Controller (PIC)

A programmable interrupt controller (PIC) is a device that manages interrupts in a computer system. It acts as a mediator between the interrupting devices and the CPU, allowing multiple devices to interrupt the CPU and allowing the CPU to prioritize and handle those interrupts in a controlled way.

In a typical computer system, the PIC is connected to the CPU and to the interrupting devices via a set of interrupt lines. When an interrupting device wants to send an interrupt to the CPU, it sends a signal on one of the interrupt lines. The PIC receives the signal and sends a corresponding interrupt request to the CPU.

The PIC also has a priority scheme that determines the order in which it sends interrupt requests to the CPU. This allows the CPU to prioritize interrupts based on the importance of the devices or events generating the interrupts.

PICs are often used in multi-core systems, where there is more than one CPU and each CPU has its own PIC. In these systems, the PICs work together to distribute interrupts evenly across the CPU cores.

PICs are often implemented in hardware, but they can also be implemented in software. Some modern operating systems, such as Linux, have software-based PICs that are implemented as part of the operating system kernel.

13.1 Deferring work

In Linux, "*deferring work*" refers to the process of delaying the execution of a task until a later time. This is often done when a task cannot be completed immediately due to external factors, such as the availability of a resource or the completion of another task.

Originally every interrupt management was divided in two levels:

- Top half: executes a minimal amount of work which is mandatory to later finalize the whole interrupt management
    - Works in a non-interruptible scheme
    - Schedules some deferred work (deferred functions)
- Bottom half: finalizes the work by deferred functions from queue and executing them. Invoked in particular reconciliation points in time.

Three ways of deferring work:

1) SoftIRQs (main mechanism, rarely used alone)
2) Tasklet

3) Work queues

SoftIRQs

In Linux, a softIRQ (short for "software interrupt") is a type of interrupt that is generated and handled entirely in software. SoftIRQs are used to perform low-priority tasks that do not need to be executed immediately, such as cleaning up data structures or updating statistics.

When a task generates a softIRQ, it is added to a queue of pending softIRQs. The kernel then schedules a special kernel thread, called a "bottom half," to handle the softIRQ later. The bottom half runs in the background and processes the tasks in the queue of pending softIRQs.

SoftIRQs are used to improve system performance by allowing the kernel to defer the execution of low-priority tasks until the CPU is idle. This can help to prevent delays in the execution of higher-priority tasks, such as servicing interrupts or running user programs.

SoftIRQs are statically allocated at compile time. Linux ensures that when a softIRQ is run on a CPU, it cannot be preempted on that CPU.

Tasklets

In Linux, a tasklet is a type of softIRQ (software interrupt) that is used to perform short tasks that do not need to be executed in a specific order. Tasklets are scheduled by the kernel and run in the background, allowing the kernel to defer the execution of low-priority tasks until the CPU is idle.

Tasklets are implemented as a special type of kernel function called a "tasklet function." When a tasklet is scheduled, the kernel executes the tasklet function in the context of the bottom half, a kernel thread that is responsible for processing softIRQs.

Tasklets are often used to perform tasks that require access to shared data structures, such as updating statistics or cleaning up data. They are also used to perform tasks that need to be executed at regular intervals, such as checking the status of a device or monitoring system performance.

Task queues

A task queue is a type of softIRQ is scheduled by the kernel and run in the background, allowing the kernel to defer the execution of low-priority tasks until the CPU is idle.

Task queues are implemented as a special type of kernel data structure that holds a list of tasks to be executed. When a task is added to a task queue, it is placed in a specific position in the list based on its priority. The kernel then schedules a special kernel thread, called a "bottom half," to execute the tasks in the task queue.

They are used to perform the same operations as above.

13.2 Timers

System timer

In Linux, the ***system timer*** is a hardware timer that is used to generate interrupts at regular intervals. The system timer is typically implemented as part of the motherboard or as a separate device and is used to provide a basic timing reference for the operating system and other system components.

The system timer is usually driven by a crystal oscillator, which generates a precise frequency that is used to determine the interval between interrupts. The interval between interrupts is usually configurable and is often set to a value of 1 millisecond or less.

The system timer is used for a variety of purposes in Linux, including:

- tracking the current time
- generating regular interrupts
- managing sleep states

Dynamic timer

Dynamic timers are implemented as kernel data structures and are used to perform tasks in the background, such as cleaning up data structures or updating statistics.

Dynamic timers are called "dynamic" because they can be created and destroyed dynamically at runtime, as needed. This allows the kernel to add and remove timers as needed, without the need to pre-allocate a fixed number of timers.

Dynamic timers are implemented using a data structure called a "*timer_list*," which contains a call-back function and a time delay. When a dynamic timer is started, the kernel schedules the call-back function to be executed after the specified time delay has expired.

Dynamic timers are often used to schedule tasks that need to be executed at a specific time or at regular intervals, such as checking the status of a device or updating statistics. They are also used to perform tasks that need to be executed in the background, such as cleaning up data structures or monitoring system performance.

14. Linux Device management

14.1 Introduction

In Linux, devices are managed using a layered approach, with each layer responsible for a different aspect of device management. The main layers of the device management stack in Linux are:

- Hardware: This layer consists of the physical devices and their associated hardware components, such as buses, ports, and connectors
- Device drivers: This layer consists of software components that communicate with the hardware and provide a standard interface for accessing the device. Device drivers are responsible for controlling the device, performing I/O operations, and managing device-specific resources
- Device files: This layer consists of special files in the file system that represent the devices. Device files are used to access the devices and perform I/O operations on them
- User space: This layer consists of user-level programs that interact with the devices through the device files

Devices are typically separated into different categories based on their type and function. The main categories of devices in Linux are:

- *block devices*: they are devices that store data in fixed-size blocks, such as hard drives, SSDs, and USB drives. Block devices are accessed through device files in the /dev/ directory, and are typically used to store and retrieve data
- *character devices*: they are devices that transmit data one character at a time, such as serial ports and terminal devices. Character devices are accessed through device files in the /dev/ directory, and are typically used to transmit data to and from the device

In Linux, devfs, sysfs, and udev are three different systems that are used to manage devices and device drivers.

- **devfs** (short for "device file system") is a virtual file system that was used in older versions of Linux to manage device files. Devfs was used to create and manage device files in the file system, and was responsible for creating and deleting device files as devices were added and removed from the system.
- **sysfs** (short for "system file system") is a virtual file system that is used to expose information about the devices and drivers in the system to user space programs. Sysfs is used to expose device attributes, such as device properties and driver information, as files in the file system, which can be accessed by user space programs through the sysfs interface.
- **udev** (short for "userspace device manager") is a system that is used to manage devices and device drivers in the Linux kernel. Udev is responsible for detecting and enumerating devices, allocating resources for devices, and creating and deleting device files in the file system. Udev uses rules and scripts to control how devices are managed and can be configured to perform custom actions when a device is added or removed from the system.

Memory mapped vs Port mapped I/O

In Linux, memory-mapped I/O (Input/Output) and port-mapped I/O are two different techniques that are used to perform I/O operations on devices.

Memory-mapped I/O involves mapping a device's memory space into the memory space of the CPU, so that the device can be accessed using memory access instructions. This allows the device to be treated like any other memory location and allows the CPU to perform I/O operations on the device using standard memory access instructions. Memory-mapped I/O is typically used for devices that have a large amount of memory, such as graphics cards or network adapters.

Port-mapped I/O involves accessing a device through a dedicated set of I/O ports, which are special memory locations that are used to communicate with the device. Port-mapped I/O is typically used for devices that have a small amount of memory, such as serial ports or terminal devices.

Character drivers and block drivers

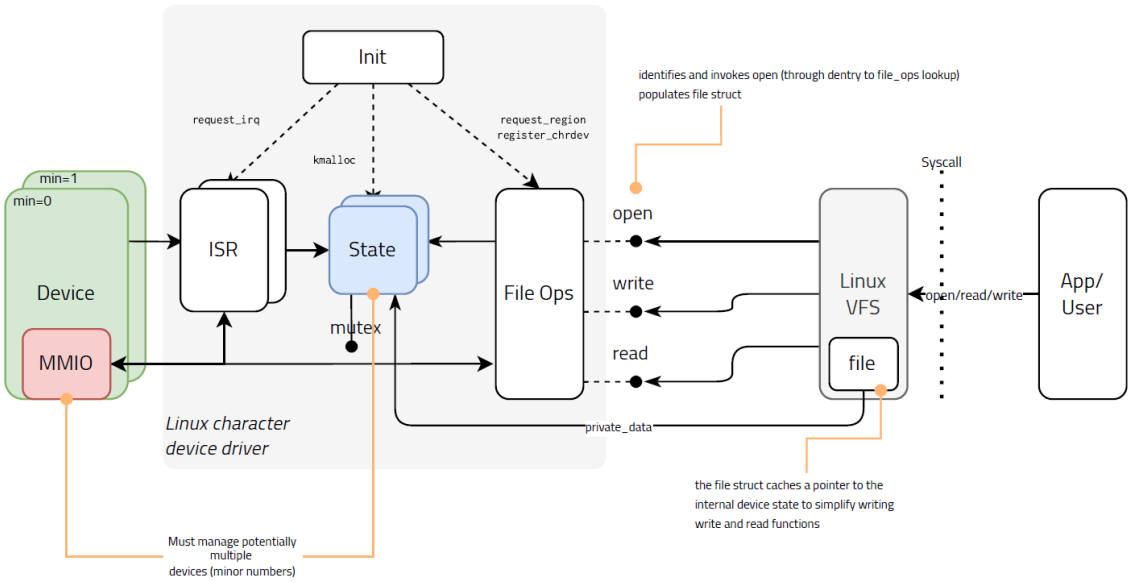
Character drivers are used to control devices that transmit data one character at a time, such as serial ports and terminal devices. Character drivers are typically accessed through device files in the /dev/ directory, and are used to transmit data to and from the device.

Block drivers are used to control devices that store data in fixed-size blocks, such as hard drives, SSDs, and USB drives. Block drivers are typically accessed through device files in the /dev/ directory, and are used to store and retrieve data from the device.

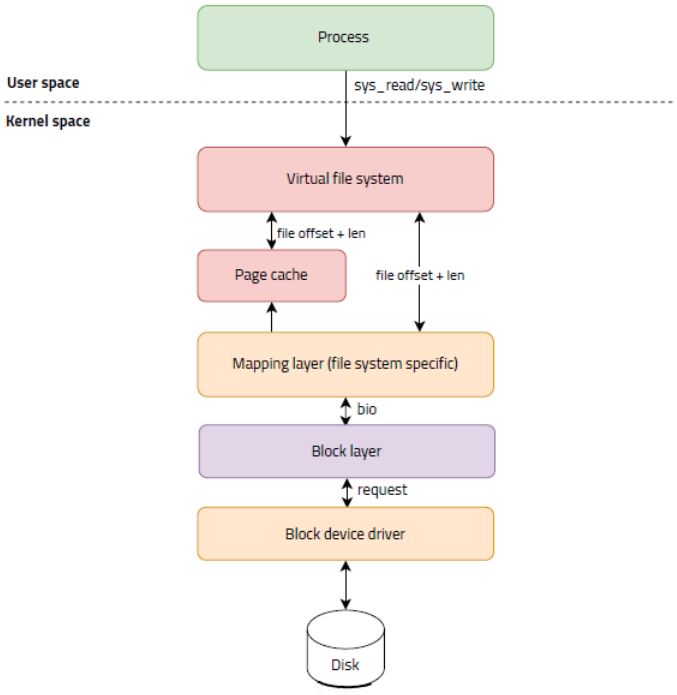
Both character drivers and block drivers are implemented as kernel modules and are loaded into the kernel at runtime. They provide a standard interface for accessing the device, and are responsible for controlling the device, performing I/O operations, and managing device-specific resources.

14.2 Low level device management

Char devices



Block devices



VFS is a module that handles two different aspects:

- it verifies if the data is already in memory (that is, resides in the kernel's page cache)
- if not, it sends the request for data to the mapping layer. The mapping layer accesses the file descriptor and pieces together the logical-to-physical mapping, therefore getting, at the end, the position of the actual disk blocks. At the end, it creates a request to the block layer through a structure called bio (block I/O)
- The block I/O layer tries to merge the list of bios into the least amount of actual requests to the driver

There is also a page cache mechanism that helps to handle requests. It is implemented as a red-black tree that, for each offset in the file, tell us what page contains that offset of the file. Typically, a page contains multiple blocks which are a multiple of a disk sector (512b).

If the page cache does not contain the data, the mapping layer

- identifies the segments of a page that correspond to contiguous sectors on disk. Internally the mapping layer works with multiples of sectors called blocks; for simplicity assume that 1 sector is 1 block.
- collects requests for `segments` that map to contiguous sectors on disk in a structure called bio. A bio references back to the original segments through a `bio_vec` pointer and contains data to iterate over it.

How is a block request managed?
Once created, one or more bios are sent to the block layer which will create the actual request to be sent to the device driver.
Before sending it, the block layer tries to merge several bio requests into single requests whenever possible. To do this, it uses a staging (software) request queue.

- Initially, the generic block layer creates a request including just one bio.
- Later, the I/O scheduler may “extend” the request either by adding a new segment to the original bio, or by linking another bio structure into the request. This is possible when the new data is physically adjacent to the data already in the request.

After a while, requests are moved from the staging queue to the actual hardware queue that will be read by the device driver and Once in a while, the `queue_rq` is called by the block layer and the device can fetch requests from the hardware queue and execute them.
Driver's `queue_rq` is invoked through a mechanism, called plugging, that adjusts the rate at which requests are dispatched to the device driver: Under a low load, operations to the driver are delayed allowing the block layer to perform more merges.

14.3 Block devices – IO schedulers

An ***I/O scheduler*** is a kernel component that controls the order in which block I/O requests are submitted to a storage device. The goal of the I/O scheduler is to improve the overall performance of the system by optimizing the way in which I/O requests are handled.

There are several different I/O schedulers available in Linux, each with its own specific set of algorithms and strategies for scheduling I/O requests. Some of the most used I/O schedulers in Linux include:

- BFQ (Budget Fair Queuing): Assigns an I/O budget to each process. Once a process is selected, it has exclusive access to the storage device until it has transferred its budgeted number of sectors. BFQ tries to preserve fairness overall, so a process getting a smaller budget now will get another turn at the drive sooner than a process that was given a large budget.
- NOOP (No Operation): this scheduler simply passes all I/O requests through to the storage device without any additional processing or sorting. NOOP is typically used with devices that have their own built-in I/O scheduling algorithms, such as some types of solid-state drives.
- DEADLINE: this scheduler uses a first-in, first-out (FIFO) queue to schedule I/O requests, but it also assigns each request a deadline by which it must be completed. This helps to ensure that I/O-intensive tasks are completed in a timely manner, while still allowing other processes to access the storage device.
- KYBER: this is a newer I/O scheduler that aims to strike a balance between high I/O throughput and low latency. It uses a prediction-based algorithm to try and anticipate the I/O needs of different processes and adjust its scheduling accordingly.

An extra scheduler is *Anticipatory scheduler*. The goal is to improve Deadline I/O applying an anticipation heuristic. When a READ request is submitted, the scheduler waits for a few milliseconds (6ms by default), speculating on the chance of receiving subsequent adjacent requests Assumption is that waiting time is compensated by the saved seeking time (typically higher).

EXTRA

Secure boot

Secure boot is a security feature that is designed to ensure that a device will only boot using trusted software. The goal of secure boot is to prevent malicious software or firmware from running on a device, which can help to protect against a variety of threats, including viruses, rootkits, and other types of malware.

Secure boot works by using a set of cryptographic keys to authenticate the software that is allowed to run on a device. These keys are stored in a hardware component called a trusted platform module (TPM), which is responsible for verifying the authenticity of the software before it is allowed to run.

When a device is turned on, the boot process begins with the BIOS or UEFI firmware. The firmware checks the signature of the bootloader, which is the software responsible for loading the operating system. If the bootloader's signature is not recognized as trusted, the device will not boot. If the bootloader's signature is recognized as trusted, it is allowed to run, and it in turn checks the signature of the operating system kernel. This process continues until the operating system is fully loaded and running.

In order for secure boot to work, a chain of trust must be established from the BIOS or UEFI firmware all the way up to the operating system. This requires that the keys used to sign each component of the boot process be properly configured and that the components themselves have not been tampered with. Secure boot is typically used in conjunction with other security measures, such as encrypted file systems and strong passwords, to provide a comprehensive security solution for devices.